Session Types for ABS

Technical Report: TUD-CS-2016-0179
Last Revision: 29.07.2016
Eduard Kamburjan

TECHNISCHE UNIVERSITÄT DARMSTADT

Software Engineering Group
Department of Computer Science

Contents

1. Introduction **3**

2. ABS **7**
2.1. Syntax and Semantics of ABS 7
2.2. ABSDL 17
2.3. Calculus 22

3. Session Types **26**
3.1. Global Types 27
3.2. Local Types 31
3.3. Projection 35
3.3.1. Projecting Global to Object-Local Types 36
3.3.2. Projecting Object-Local to Method-Local Types 38
3.4. Translation of Types to Regular Expressions 41
3.4.1. Translation of Global Types to Regular Expressions 41
3.4.2. Translation of Local Types into Regular Expressions 42
3.5. Well-Formedness 43

4. Verification **46**
4.1. Admissibility 46
4.2. Translation of Method-Local Types into ABSDL 50
4.3. Scheduling with Session Automata 57

5. Conclusion **61**

A. Appendix **63**
A.1. Proofs 63

Bibliography **73**

1. Introduction

Motivation

The abstract behavioral specification (ABS) language is an object-oriented language, designed to model distributed systems. The concurrency model of ABS is based on invariants and compositional reasoning; reasoning about a concurrent system is possible by *invariants of objects*: Objects must ensure that their invariant holds at every moment the processor switches the active process. The activated process then can rely on this guarantee.

Another aspect of the concurrency model of ABS is *cooperative scheduling*: An object can only switch the active process if the currently active process explicitly releases control or terminates. Thus processes running in the same object can not be interleaved arbitrarily, but only as programmed by the developer.

Such invariants for ABS can be verified with the automatic theorem prover KeY-ABS, which encodes symbolic execution in a dynamic logic. Invariants are specified directly in this logic and verified to hold at the release points of all methods in a class. There is no automatic reasoning about the specification from a global point of view. Especially it is not possible to specify the *global communication pattern* of a system. The communication pattern of a system run is the sequence of executed communication events, such as sending and receiving.

Multi-party session types for asynchronous systems [18] are an established type discipline to specify and verify distributed systems. Session types allow to *specify globally*, but to *verify locally*. I.e. a global specification can be verified with only the code of single endpoints involved in the communication. Former work was mainly concerned with session types for concurrency models based on channels.

In this work we adapt the session type approach to ABS. We design a session type specification language for the concurrency model of ABS, which is not based on channels and in which executions of endpoints can not be interleaved arbitrarily.

Introduction

We provide a specification language which is

1. verifiable in the KeY-ABS theorem prover and

2. limits reasoning about side effects and the heap memory to a minimum.

KeY-ABS has uniform classes, all methods are available at any time. We do not specify or verify typestate [28] by introducing additional fields to keep track of the current point in the protocol. We present two approaches:

- A static check whether it suffices to demand that all processes follow their local specification to show that they are composed correctly In this case we give no guarantee how the system continues if one process does not follow its specification.

- Additionally to the derivation of specifications for processes, we derive a specification on object-scheduler. If the object schedules processes according to this specification, the order is guaranteed.

The concurrency model of ABS differs from the concurrency model of π-calculus based systems by using *futures* instead of channels and by grouping processes by the object which are executing them. Thus the session types for ABS need additional concepts for the following three points:

1. **Correct usage of futures:** ABS demands that every communication between two objects is realized by an asynchronous method call. Each such method call uses a new future as a place holder on the caller side and starts a new process on the callee side. The future transports additional data strictly in the following order:
   - The caller writes the method parameters
   - The callee reads the method parameters
   - The callee writes the return value (resolving the future)
   - Anyone may read the return value (fetching the future)

   In contrast to channels it is not possible to send arbitrary data at arbitrary points of time from any endpoint who has access to the future. Also the callee process does not have explicit access to the future it computes.

We use more session type actions than channel-based session types, because there are more operations which can be performed: additionally to sending and receiving a method call, we model the resolving and fetching of a future to ensure that the session type describes a communication that is permissible in the concurrency model with futures.

2. **Communication of choice:** In classical session types, if an endpoint chooses a branch of execution it sends a branch-label over a channel to any endpoint that must be notified about the choice. This is not practical in the ABS model, in which every sending either creates a new process (if communicated via a method call) or prevents the choosing process from sending another value (if communicated via the return value).

We distinguish between forward choices, which are communicated from the caller to the callee, and backward choices, which are communicated from the callee to the caller. We use different mechanisms to handle these.

- In *forward choices*, the choice is communicated via the method-name, i.e. every branch is identified by the method-name instead of a branch-label. The callee can resume its execution depending on which method has been called. As we exclude reasoning about the heap, already active processes in the callee object are not notified about the choice and can not make their continuation depend on the choice. The reason is that processes in the same object can only communicate via the heap.

- In *backward choices*, the choice is communicated via a constructor of an algebraic data type, i.e. the return type is an algebraic data type and the branch is identified by the outermost constructor. Only the caller process is notified about the change, other active processes in the same object must continue it the same way in every branch. The return value must not be an algebraic data type for methods that are not communicating a backward choice.

3. **Cooperative Scheduling** Processes in ABS are grouped by the executing object and can not be interleaved arbitrarily. Instead, a process is executed until it explicitly releases control by suspending or terminating itself.

We use an additional session type action, which is not needed in models for channel-based session types, to model release of control in the session type. This action ensures that the processes can be interleaved as specified.

Scheduling in ABS provides no guarantees like fairness. It also does not specify which process will be reactivated if several processes can be reactivated. Our specification language does not rely on a specific scheduling algorithm: at each point the scheduler must decide what process to start, it is guaranteed that there is only one possible process.

To ensure that our specification can be verified with KeY-ABS without dynamic checks at runtime, we impose the following assumptions and restrictions:

- Our session types are not able to specify the usage of fields with future types. Reading them results in a communication history which is not captured by any session type.
- The guard of an **await** statement can only be a future. This enables us to derive when a process is reactivated. Processes may reactivate after a side-effect of another process in the same object, if arbitrary guards are allowed.

  Example 1
  Consider a server which has an initializing process, which suspends itself after initializing the fields and is reactivated once 2 accesses have been served. The processes computing these accesses can communicate this by increasing a counter field.

  To verify such a specification, it must be verified that the access-processes indeed increases the counter. We are concerned with communication patterns and do not allow such specifications. This way we are able to reduce reasoning about the heap memory.
- All objects are already created and all endpoints who communicate in the session have pointers to each other. Creation of new objects and their propagation would complicate the specification without any further insights, as we already deal with propagation for futures. This assumption simplifies the presentation.
- Every method has at most one session type. In general, methods may realize different communication patterns, depending on the inner state of their object. This could be reflected by different communication at different positions in the type. To enable this, one must verify that the right communication pattern is realized at a given precondition. This would again require reasoning about side-effects (the state must be changed correctly before the second call) and the heap memory.
- Each cog has at most one object. This is a current restriction of the KeY-ABS prover.
- Each repetition is iterated only finitely often. As the KeY-ABS prover currently only allows reasoning about partial correctness, we are only able to verify finite behavior.

Notation

Sets and Sequences We denote the powerset of a set A with $\mathcal{P}(A) = \{A' \mid A' \subseteq A\}$ and the set of finite sequences of elements of A with A^*. We write $[a, b, c, \dots]$ for the sequence containing $a, b, c, \dots$ in this order. Given a sequence $S \in A^*, i \in \mathbb{N}, a \in A$, we write $|S|$ for the length of S, $S[i]$ for the ith element in s and $S \setminus a$ for the sequence that is obtained by the usual removing operation of every occurrence of a in S and $\mathit{last}(S)$ for the last element of S. Given two sequences S, S' we write $\pi(S, S')$ if S is a permutation of S' and $S \circ S'$ for the concatenation of S and S'. The empty sequence is denoted ϵ. We denote that S is a prefix of S' by $S \sqsubseteq S'$.

Functions Given two sets A, B, we write $A \to B$ for the set of all total functions from A to B and $A \rightharpoonup B$ for the set of all partial functions from A to B. We write $\bot$ for undefined and $f(x) = \bot$ if f is undefined for x. We denote the domain of a partial function with $\mathbf{dom}(f) = \{x \in A \mid f(x) \neq \bot\}$ and its image with $\mathbf{im}(f) = \{x \in B \mid \exists a \in A.\ f(a) = x\}$. Given a function $f : A \to B$ and two elements $a \in A, b \in B$ we write $f[a \mapsto b]$ for the updated function which is defined by

$$f[a \mapsto b](x) = \begin{cases} b & \text{if } x = a \\ f(x) & \text{otherwise} \end{cases}$$

2. ABS

ABS [21] is designed to model concurrent systems in a way that simplifies reasoning about their behavior. Communication between two objects is only possible by asynchronous method calls and reading the return value. There are no public fields. An asynchronous method call is realized by establishing the connection via a unique future. This future is a place holder for the caller, who can wait until the called method terminates and then read from the future. If the caller attempts to read while the called method is not yet terminated, the process blocks. If the process releases control and waits for the termination of the callee, the object can execute another process. This is the only way to switch the active process, an object can not interleave its processes arbitrarily.

An ABS system consists of multiple objects, each with a set of executable methods. Each object has an active process and may have several *suspended* processes. If a method of the object is called while a process is already active, the execution of the new process for this method call is also suspended. A suspended process which is waiting for a future can only continue once the future it is waiting for is *resolved*, i.e. the method call it is handling is terminated.

In this chapter we describe a subset of ABS. As we are only concerned with communication and not computation, we have a more simple model of expressions. Besides that, we only allow suspension on futures.

2.1. Syntax and Semantics of ABS

Definition 1 (Data Type System)
An ABS-data-type system is a triple $(\mathsf{A}, \mathsf{C}, \mathsf{I})$, where

- I is the partially ordered set of all interface names, where $i \preceq i'$ denotes that i extends i'.
- C is the set of all class names, where $c \preceq i$ denotes that class c implements interface i.
- A is the set of all algebraic data types. Each algebraic data type a is represented as a tuple $(id, constr, ar, type)$ where id is the name, $constr$ the set of constructor names, $ar : constr \to \mathbb{N}$ maps constructors to their arity and $type : constr \to (\mathsf{I} \cup \mathsf{A})^*$ maps constructors to their types. We refer to an algebraic data type by its unique name.

We assume that all data types have unique names, $type$ respects the arity of its argument and all sets of constructors are disjoint. Given an algebraic data type $a = (id, constr, ar, type)$, we denote its set of constructor names with $constr(a)$. If a data type system $(\mathsf{A}, \mathsf{C}, \mathsf{I})$ is fixed, we write Con for $\bigcup_{a \in \mathsf{A}} constr(a)$, the set of all constructor names.

We demand that there are pre-defined algebraic data types **bool** with two 0-arity constructors **true** and **false**, **Int** with one 0-arity constructor for each number $n \in \mathbb{N}$, and **Exception** with one 0-arity constructor **PatternFailure**. The **Exception** data type may have more user-defined constructors.

We follow mostly the subset presented in [16] and only give the definition of syntax for methods, assuming the data type system is given when the session type is specified. We denote the set of all fields with Fl and the set of all variables with V.

Definition 2 (Syntax of ABS Statements)
Let C range over all constructors in a given ABS-data-type system, x over local variables and fields, fl over fields, f over futures Fut, m over methods Met and T over interface names and algebraic data type names.

The notation $\vec{\cdot}$ denotes a sequence of elements, of fitting arity.

Expressions e are defined by the grammar

$$e ::= e\&e \mid !e \mid C(\vec{e}) \mid x \mid e + e \mid e * e \mid -e$$

Branches b and statements s are defined by the grammar

$$\begin{aligned} b ::= & C(\vec{e}) => s \mid \textbf{default} => (s \mid \epsilon) \\ s ::= & s; s \mid \textbf{await}\ f \mid x = e \mid Tv = C(\vec{e}) \mid Fut\langle T\rangle f = fl!m(\vec{e}) \mid \textbf{return}\ e \mid \\ & \textbf{if}(e)\textbf{then}\{s\}\textbf{else}\{s\} \mid \textbf{while}(e)\{s\} \mid \textbf{throw}\ e \mid \textbf{try}\{s\}\textbf{catch}(e)\big[b\big]^{+} \mid \textbf{case}(e)\big[b\big]^{+} \end{aligned}$$

The statement $Fut\langle T\rangle f = fl!m(\vec{e})$ executes an asynchronous method call on the method m on the object stored in fl. The return type of m must be T and f is the future used to handle this call. We refrain from giving a formal type system, but assume one similar to [21], where matching is also described in detail.

The **case** statement is a pattern matching statement, which matches the expression e against the guard $c(\vec{e})$ of all branches. If e can be matched against a guard, the branch is executed. If there is no matching branch, but a **default** branch is provided, then the **default** branch is executed. If no **default** branch is provided, then a **PatternFailure** exception is thrown. The **try** statement has a catching part **catch** which matches a thrown exception against all the provided branches. The default branching is handled by matching with a wild-card _ in ABS, we use **default** to simplify semantics. The other statements are standard in imperative and object-oriented languages.

Definition 3 (Method)
A method m is a tuple $(id, n, v, type, s)$, where id is the name, n the number of parameters, $v \in V^n$ the names of the parameters (which we treat as special variables), the function $type : [1, \ldots, n] \to (\mathsf{I} \cup \mathsf{A})$ maps parameters to their type and s is a statement of the form $s'; \textbf{return}\ e$ where s' contains no other **return**. We refer to a method by its unique name and to its return type by $ret(m)$. We denote the set of all method names by Met.

Example 2
Consider the following method: `run` is a client which sends 10 messages to a server stored in the field `server` (1). It waits until the server resolves the future (2), reads its value (3) and counts how often the request succeed or not. In case an exception is thrown upon reading from the future, the client sets its counter to 10 and thus stops the loop from further iterations (4). The example assumes there are fields `server`, `success` and `fail`, there is an abstract data type Answer with constructors Ok and Deny and that the method request return a value of type Answer.

```
Unit run(){
    Int i = 0;
    while(i < 11){
        Fut<Answer> f = server!request();
        await f?
        try{
            Answer a = f.get;
            case(a){
                Ok => success = success + 1;
                Deny => fail = fail + 1;
            }
        } catch (exc) {
            default => i = 10;
        }
        i = i+1;
    }
}
```

The communication history of a process (object or system) is a sequence of communication events executed by this process (object or system) in the past. Each communication event describes a visible operation on a future: An event is added if a process starts computing a future, a process issues a computation by a method call, a process terminates and resolves its future or a process releases control.

Definition 4 (Communication Events)
The set of all events Ev, is defined by the following grammar. Let $\mathsf{ev} \in \mathsf{Ev}$.
Let O, O' range over the set of all object *names* Ob, f, f' over Fut, m over Met, e over expressions and $\vec{e}$ over lists of expressions.

$$\begin{aligned}\mathsf{ev} ::=& \mathsf{invEv}(O, O', f, m, \vec{e}) \mid \mathsf{invREv}(O, O', f, m, \vec{e}) \mid \mathsf{futEv}(O, f, m, e) \mid \mathsf{futREv}(O, f, e) \\ &\mid \mathsf{awaitEv}(O, f, f') \mid \mathsf{throwEv}(O, f, m, e) \mid \mathsf{throwREv}(O, f, e)\end{aligned}$$

The events model the following communication

- An invocation event $\mathsf{invEv}(O, O', f, m, \vec{e})$ models that O calls the method m of object O', passes $\vec{e}$ as the parameter and uses f as a handle. This does not model that O' already received or handled the call.
- An invocation reaction event $\mathsf{invREv}(O, O', f, m, \vec{e})$ models that the object O' starts the execution of method m, which was issued by object O with parameters $\vec{e}$ and will resolve the future f when terminating.
- A resolving event $\mathsf{futEv}(O, f, m, e)$ models that object O resolves f by finishing the execution of method m. Future f now contains e.
- A fetching event $\mathsf{futREv}(O, f, e)$ models that object O reads from future f the value e.
- A suspending event $\mathsf{awaitEv}(O, f, f')$ models that object O suspends the computation of future f until future f' is resolved.

- A throwing event $\mathsf{throwEv}(O, f, m, e)$ models that object O resolves f by throwing an exception while executing method m. Future f does not contain the thrown exception e.
- A catching event $\mathsf{throwREv}(O, f, e)$ models that object O catches exception e when reading from future f.

  A *history* is a sequence of events.

Example 3
One iteration of the code in Example 2 can produce the following history from the view of O, if executed from an object O and a process computing f_0:

$$[\mathsf{invEv}(O, O', f, \mathsf{req}, 4), \mathsf{awaitEv}(O, f_0, f), \mathsf{futREv}(O, f, \mathsf{Ok})]$$

Not all histories describe a possible sequence of communication events. For example, each invocation event may have at most one corresponding invocation reaction event and each invocation event must use a fresh future.

Histories that can be generated by an ABS system are *well-formed*.

Definition 5 (Well-Formed Histories)
Let h be a history. h is well-formed if the following holds for every $ieq|h|$. For readability we refrain from explicitly binding all variables in the conditions. All free variables are implicitly existentially bound.

- If $h[i]$ is an invocation event, then its future does not occur in $h[1..i-1]$.

  $$h[i] = \mathsf{invEv}(O, O', f, m, e) \rightarrow f \notin \mathit{futures}(h[1..i-1])$$

  Where $\mathit{futures}(h)$ is the set of all futures occurring somewhere in the history h.

- If $h[i]$ is an invocation reaction event, then there is exactly one corresponding invocation event and no other invocation reaction event on the same future in $h[1..i-1]$ and the objects are not identical.

  $$\begin{aligned} h[i] = \mathsf{invREv}(O, O', f, m, e) \rightarrow& \\ &\exists j < i.\ h[j] = \mathsf{invEv}(O, O', f, m, e) \wedge \\ &\quad \big(\forall j < k < i.\ h[k] \neq \mathsf{invREv}(O, O', f, m, e)\big) \wedge O \neq O' \end{aligned}$$

- If $h[i]$ is a resolving event, then there is a corresponding invocation reaction event and there is no other resolving event and no throwing event on the same future in $h[1..i-1]$

  $$\begin{aligned} h[i] =& \mathsf{futEv}(O, f, m, e) \rightarrow \exists j < i.\ h[j] = \mathsf{invREv}(O, O', f, m, e) \wedge \\ &\forall k < i.\ h[k] \neq \mathsf{futEv}(O, f, m, e) \wedge h[k] \neq \mathsf{throwEv}(O, f, m, e) \end{aligned}$$

- If $h[i]$ is a throwing event, then there is a corresponding receiving event and there is no other throwing event and no resolving on the same future in $h[1..i-1]$

  $$\begin{aligned} h[i] =& \mathsf{throwEv}(O, f, m, e) \rightarrow \exists j < i.\ h[j] = \mathsf{invREv}(O, O', f, m, e) \wedge \\ &\forall k < i.\ h[k] \neq \mathsf{futEv}(O, f, m, e) \wedge h[k] \neq \mathsf{throwEv}(O, f, m, e) \end{aligned}$$

- If $h[i]$ is a fetching event, then there is a corresponding resolving event in $h[1..i-1]$

$$h[i] = \mathsf{futREv}(O, f, e) \to \exists j < i.\ h[j] = \mathsf{futEv}(O', f, m, e)$$

- If $h[i]$ is a catching event, then there is a corresponding throwing event in $h[1..i-1]$

$$h[i] = \mathsf{throwREv}(O, f, e) \to \exists j < i.\ h[j] = \mathsf{throwEv}(O', f, m, e)$$

- If $h[i]$ is a suspending event, then there is a corresponding invocation reaction event in $h[1..i-1]$ and no resolving event on the same future in between. Also the future which is waited for is invoked.

$$\begin{aligned} h[i] =& \mathsf{awaitEv}(O, f, f') \to \\ & \exists j < i.\ \Big(h[j] = \mathsf{invREv}(O', O, f, m, e) \land \\ & \qquad \forall j < k < i.\ \big(h[k] \neq \mathsf{futEv}(O, f, m, e) \land h[k] \neq \mathsf{throwEv}(O, f, m, e)\big)\Big) \\ & \land \exists j < i.\ h[j] = \mathsf{invEv}(O'', O''', f', m', e') \end{aligned}$$

Example 4
Consider the history from Example 3:

$$[\mathsf{invEv}(O, O', f, \mathsf{req}, 4), \mathsf{awaitEv}(O, f_0, f), \mathsf{futREv}(O, f, \mathsf{Ok})]$$

It is not well-formed, because it only describes the part of O – the future f is read without being resolved first. A well-formed history is:

$$[\mathsf{invEv}(O, O', f, \mathsf{req}, 4), \mathsf{awaitEv}(O, f_0, f), \mathsf{invREv}(O, O', f, \mathsf{req}, f), \mathsf{futEv}(O', f, \mathsf{req}, \mathsf{Ok}), \mathsf{futREv}(O, f, \mathsf{Ok})]$$

Futures identify a communication. Thus when renaming a future *consistently*, i.e. renaming it in every event in a history where it occurs, does not change the communication pattern. We say that two histories are *future-equivalent* if one can be transformed into the other by renaming futures.

Definition 6 (Future-Equivalence)
Two well-formed histories s, s' are *equivalent up to futures*, if s' is equal to s after renaming all futures which are introduced within s'. A future is introduced by an invocation or an invocation reaction event. Let $\{f \mid \exists i \leq |s|.\ s[i] = \mathsf{invEv}(O, O', f, m, e) \lor \mathsf{invREv}(O, O', f, m, e)\} = \{f_1, \dots, f_n\} = F(s)$ be the set of all futures occurring in s. Then

$$s \equiv_{\mathsf{Fut}} s' = \exists f'_1, \dots, f'_n \in \mathsf{Fut} \setminus F(s).\ s = s'[f_1 \backslash f'_1] \dots [f_n \backslash f'_n]$$

where $[f \backslash f']$ is syntactical substitution of every occurrence of f by f'.

During the execution of a process, at every step at most one single event is appended to the history. Session types require other operations on histories to specify branching and repetition. We introduce the following operators on sets of sequences as regular expression for histories.

Definition 7 (Regular Expressions of Sequences)
Let S, S' be two sets of sequences and k a natural number. We use the following operators on sets of sequences as regular expression for histories, where $\mathcal{L}$ maps such an expression to the set of sequences it describes.

- For a set S we define $\mathcal{L}(S) = S$
- $S \circ S'$ with $\mathcal{L}(S \circ S') = \{s \circ s' \mid s \in \mathcal{L}(S), s' \in \mathcal{L}(S')\}$ is componentwise concatenation
- $S + S'$ with $\mathcal{L}(S + S') = \mathcal{L}(S) \cup \mathcal{L}(S')$ is alternative
- S^k is k-bounded repetition with
  - $\mathcal{L}(S^0) = \{\epsilon\}$,
  - $\mathcal{L}(S^1) = \{s \in S \mid s \equiv_{\mathsf{Fut}} s\}$
  - $\mathcal{L}(S^k) = \mathcal{L}(S^1 \circ S^{k-1}), k > 1$
- $\mathcal{L}(S^*) = \bigcup_{k\in\mathbb{N}} S^k$ is unbounded repetition

We define the semantics of processes, objects and systems in terms of small step semantics.

Definition 8 (Process)
Let D be the set of possible values for variables and fields. A process is a tuple (s, ρ, h, f) where

- s is a statement or ϵ,
- $\rho : V \to D$ maps variables and parameters to their values,
- h is a history
- f is a future

We assume that ρ respects the type of the variable. We denote the history component of a process $p = (s, \rho, h, f)$ with $his(p) = h$. A process is *terminated* if $s = \epsilon$

The semantics of processes is defined in Figure 2.1 as the small-step semantics $(\sigma, p) \to_{O,F} (\sigma', p')$, which transforms one process p executing method m in an object with name O into another process p' by executing the next statement with a store $\sigma : Fl \to D$ and possible modifying the store into σ'. The set of futures F denotes the set of already used futures and π is a meta variable for the remaining program. The rules are standard with the exception of **suspend**: The **await** statement is not removed, only a awaitEv event is added. The object semantics are suspending a process if it adds an awaitEv event, the **await** statement is removed upon reactivation. It is used to keep track of what future a suspended process is waiting for. We denote the set of all processes with Proc.

The semantics of expressions is defined with a partial function $[\![\cdot]\!]_{\sigma,\rho}$:

$$
\begin{aligned}
[\![e_1 + e_2]\!]_{\sigma,\rho} &= [\![e_1]\!]_{\sigma,\rho} + [\![e_2]\!]_{\sigma,\rho} && \text{if } [\![e_1]\!]_{\sigma,\rho}, [\![e_2]\!]_{\sigma,\rho} \in \mathbb{N} \\
[\![e_1 * e_2]\!]_{\sigma,\rho} &= [\![e_1]\!]_{\sigma,\rho} * [\![e_2]\!]_{\sigma,\rho} && \text{if } [\![e_1]\!]_{\sigma,\rho}, [\![e_2]\!]_{\sigma,\rho} \in \mathbb{N} \\
[\![-e_1]\!]_{\sigma,\rho} &= -[\![e_1]\!]_{\sigma,\rho} && \text{if } [\![e_1]\!]_{\sigma,\rho} \in \mathbb{N} \\
[\![e_1 \& e_2]\!]_{\sigma,\rho} &= [\![e_1]\!]_{\sigma,\rho} \wedge [\![e_2]\!]_{\sigma,\rho} && \text{if } [\![e_1]\!]_{\sigma,\rho}, [\![e_2]\!]_{\sigma,\rho} \in \mathbb{B} \\
[\![!e_1]\!]_{\sigma,\rho} &= \neg[\![e_1]\!]_{\sigma,\rho} && \text{if } [\![e_1]\!]_{\sigma,\rho} \in \mathbb{B} \\
[\![v]\!]_{\sigma,\rho} &= \rho(v) \\
[\![fl]\!]_{\sigma,\rho} &= \sigma(fl) \\
[\![C(\vec{e})]\!]_{\sigma,\rho} &= C(\overrightarrow{[\![e]\!]_{\sigma,\rho}})
\end{aligned}
$$

$$\frac{}{\big(\sigma,(v = e;\pi,\rho,h,f)\big) \rightarrow_{O,F} \big(\sigma,(\pi,\rho[v \mapsto [\![e]\!]_{\sigma,\rho}],h,f)\big)}\ \textbf{assign}$$

$$\frac{}{\big(\sigma,(fl = e;\pi,\rho,h,f)\big) \rightarrow_{O,F} \big(\sigma[fl \mapsto [\![e]\!]_{\sigma,\rho}],(\pi,\rho,h,f)\big)}\ \textbf{assignField}$$

$$\frac{}{\big(\sigma,(C\ v = c(\vec{e});\pi,\rho,h,f)\big) \rightarrow_{O,F} \big(\sigma,(\pi,\rho[v \mapsto [\![c(\vec{e})]\!]_{\sigma,\rho})],h,f)\big)}\ \textbf{init}$$

$$\frac{f' \notin F \quad \rho' = \rho[fr \mapsto f']}{\big(\sigma,(Fut\langle T\rangle\ fr = fl!m(\vec{e});\pi,\rho,h,f)\big) \rightarrow_{O,F} \big(\sigma,(\pi,\rho',h \circ [\mathsf{invEv}(O,[\![fl]\!]_{\sigma,\rho},f',m,\overrightarrow{[\![e]\!]_{\sigma,\rho}})],f)\big)}\ \textbf{call}$$

$$\frac{}{\big(\sigma,(\mathbf{await}\ f';\pi,\rho,h,f)\big) \rightarrow_{O,F} \big(\sigma,(\mathbf{await}\ f';\pi,\rho,h \circ [\mathsf{awaitEv}(O,f,f')],f)\big)}\ \textbf{suspend}$$

$$\frac{}{\big(\sigma,(\mathbf{return}\ e,\rho,h,f)\big) \rightarrow_{O,F} \big(\sigma,(\epsilon,\rho,h \circ [\mathsf{futEv}(O,f,m,[\![e]\!]_{\sigma,\rho})],f)\big)}\ \textbf{return}$$

$$\frac{}{\big(\sigma,(\mathbf{throw}\ e;\pi,\rho,h,f)\big) \rightarrow_{O,F} \big(\sigma,(\epsilon,\rho,h \circ [\mathsf{throwEv}(O,f,m,[\![e]\!]_{\sigma,\rho}))],f\big)}\ \textbf{throw}$$

$$\frac{[\![e]\!]_{\sigma,\rho} = \mathbf{true}}{\big(\sigma,(\mathbf{if}(e)\mathbf{then}\{s\}\mathbf{else}\{s'\};\pi,\rho,h,f)\big) \rightarrow_{O,F} \big(\sigma,(s;\pi,\rho,h,f)\big)}\ \textbf{ifthen}$$

$$\frac{[\![e]\!]_{\sigma,\rho} = \mathbf{false}}{\big(\sigma,(\mathbf{if}(e)\mathbf{then}\{s\}\mathbf{else}\{s'\};\pi,\rho,h,f)\big) \rightarrow_{O,F} \big(\sigma,(s';\pi,\rho,h,f)\big)}\ \textbf{ifelse}$$

$$\frac{[\![e]\!]_{\sigma,\rho} = \mathbf{false}}{\big(\sigma,(\mathbf{while}(e)s;\pi,\rho,h,f)\big) \rightarrow_{O,F} \big(\sigma,(\pi,\rho,h,f)\big)}\ \textbf{whilefalse}$$

$$\frac{[\![e]\!]_{\sigma,\rho} = \mathbf{true}}{\big(\sigma,(\mathbf{while}(e)\{s\};\pi,\rho,h,f)\big) \rightarrow_{O,F} \big(\sigma,(s;\mathbf{while}(e)\{s\};\pi,\rho,h,f)\big)}\ \textbf{whiletrue}$$

$$\frac{j \in I \quad [\![e_j]\!]_{\sigma,\rho} = [\![e]\!]_{\sigma,\rho}}{\big(\sigma,(\mathbf{case}(e)(e_i \Rightarrow s_i)_{i\in I};\pi,\rho,h,f)\big) \rightarrow_{O,F} \big(\sigma,(s_j;\pi,\rho,h,f)\big)}\ \textbf{case}$$

$$\frac{\forall j \in I.\ [\![e_j]\!]_{\sigma,\rho} \neq [\![e]\!]_{\sigma,\rho} \quad \exists j \in I.\ e_i = \mathbf{default}}{\big(\sigma,(\mathbf{case}(e)(e_i \Rightarrow s_i)_{i\in I};\pi,\rho,h,f)\big) \rightarrow_{O,F} \big(\sigma,(s_i,\rho,h,f)\big)}\ \textbf{caseDefault}$$

$$\frac{\forall j \in I.\ [\![e_j]\!]_{\sigma,\rho} \neq [\![e]\!]_{\sigma,\rho} \quad \exists j \in I.\ e_i = \mathbf{default}}{\big(\sigma,(\mathbf{case}(e)(e_i \Rightarrow s'_i)_{i\in I};\pi,\rho,h,f)\big) \rightarrow_{O,F} \big(\sigma,(\mathbf{throw\ PatternFailure},\rho,h,f)\big)}\ \textbf{caseFail}$$

$$\frac{\begin{array}{c}\big(\sigma,(s,\rho,h,f)\big) \rightarrow_{O,F} \big(\sigma',(s',\rho',h',f)\big) \quad h' = h \circ s \\ s = \epsilon \vee (s = [ev] \wedge ev \neq \mathsf{throwEv}(O',f,m,e'))\end{array}}{\big(\sigma,(\mathbf{try}\{s\}\mathbf{catch}(e)\big[b\big]^{+};\pi,\rho,h,f) \rightarrow_{O,F} \big(\sigma,(\mathbf{try}\{s'\}\mathbf{catch}(e)\big[b\big]^{+};\pi,\sigma',\rho',h',f)\big)}\ \textbf{tryInner}$$

$$\frac{\big(\sigma,(s,\rho,h,f)\big) \rightarrow_{O,F} \big(\sigma',(\epsilon,\rho',h',f)\big) \quad h' = h \circ ev \quad ev = \mathsf{throwEv}(O',f,m,e')}{\big(\sigma,(\mathbf{try}\{s\}\mathbf{catch}(e)(e_i \Rightarrow s'_i)_{i\in I};\pi,\rho,h,f)\big) \rightarrow_{O,F} \big(\sigma',(\mathbf{case}(e')(e_i \Rightarrow s'_i)_{i\in I};\pi,\rho',h',f)\big)}\ \textbf{tryF}$$

$$\frac{}{\big(\sigma,(\mathbf{try}\{\epsilon\}\mathbf{catch}(e)(e_i \Rightarrow s'_i)_{i\in I});\pi,\rho,h,f)\big) \rightarrow_{O,F} \big(\sigma,(\pi,\rho,h,f)\big)}\ \textbf{tryEnd}$$

All free variables are implicitly quantified with existantial quantifiers.

Figure 2.1.: Small-Step Semantics for ABS Processes

We define the starting of a new method with the following auxiliary function:

Definition 9 (Process Initialisation)
Given a function s that maps method names to the method body belonging to this name, an event $\mathsf{invEv}(O, O', f, m, \vec{e})$ initilizes a process as follows:

$$\mathfrak{P}(\mathsf{invEv}(O, O', f, m, \vec{e})) = (s(m), \rho, f, [\mathsf{invREv}(O, O', f, m, \vec{e})])$$

where $\rho(v_i) = e_i$ holds for each e_i in $\vec{e}$.

Each object has several processes, but only one can be active at a time. We refer to the future an active process is computing as the active future. We say that a process releases or suspends control when the process becomes deactivated without terminating. Processes share the *heap* memory. When a process stops being executed and is continued later, the process continues with the current heap memory of the object, not the state when the process returned control. Also whenever a process appends events to its history, the events are also appended to the objects history.

Definition 10 (Object)
An object is a tuple (O, P, σ, p, h) where

- $O \in \mathsf{Ob}$ is the name,
- $P : \mathbb{N} \rightharpoonup \mathsf{Proc} \setminus \{p\}$ maps numbers to suspended processes,
- $\sigma : Fl \to D$ maps fields to their values,
- $p \in \mathsf{Proc}$ is the active process and
- h is a history

We identify an object with its name, but write $id(o)$ for the name of an object if we want to distinguish name and object explicitly.

An object (O, P, σ, p, h) is terminated if $p = \bot$ and $\mathbf{dom}(P) = \emptyset$, i.e. no object is active and none is suspended. The semantics of objects are defined with the following rules, where free variables are implicitly existentially bound:

$$\frac{(\sigma, p) \to_{O,F} (\sigma', p') \quad his(p) = his(p')}{(O, P, \sigma, p, h) \to_F (O, P, \sigma', p', h)}\ \textbf{inner}$$

$$\frac{\begin{array}{c}(\sigma, p) \to_{O,F} (\sigma', p') \quad his(p') = his(p) \circ [ev] \quad h' = h \circ [ev] \\ ev = \mathsf{throwEv}(O, f, m, e) \vee ev = \mathsf{futEv}(O, f, m, e)\end{array}}{(O, P, \sigma, p, h) \to_F (O, P, \sigma', \bot, h')}\ \textbf{terminate}$$

$$\frac{\begin{array}{c}(\sigma, p) \to_{O,F} (\sigma', p') \quad his(p') = his(p) \circ [ev] \quad h' = h \circ [ev] \\ ev = \mathsf{awaitEv}(O, f, f') \qquad i \notin \mathbf{dom}(P)\end{array}}{(O, P, \sigma, p, h) \to_F (O, P[i \mapsto p'], \sigma', \bot, h')}\ \textbf{suspend}$$

$$\frac{\begin{array}{c}(\sigma, p) \to_{O,F} (\sigma', p') \qquad his(p') = his(p) \circ [ev] \qquad h' = h \circ [ev] \\ ev \neq \mathsf{throwEv}(O, f, m, e)) \quad ev \neq \mathsf{futEv}(O, f, m, e) \quad ev \neq \mathsf{awaitEv}(O, f, f')\end{array}}{(O, P, \sigma, p, h) \to_F (O, P, \sigma', p', h')}\ \textbf{communicate}$$

The rule **inner** handles the case where the active process executes a statement without adding an event to its local history. The rule **terminate** handles termination and **suspend** suspension. These rules are triggered by the added event, not by comparing the code of the process. Finally **communicate** handles the case where an event is added, but does not alter the active process.

The active future of an object, is the future which is computed by the active process.

Definition 11 (Active Future)
Let $o = (O, P, \sigma, p, h)$ be an object with $p = (s, \rho, h, f)$. The active future of O is the future of p and we write $active(o) = f$.

The history of a system S is the behavior of the whole system. To analyze the behavior in the history from a local point of view, we introduce a projection on objects. The projection describes the behavior of a single object, i.e. those events which are issued by a given object.

Definition 12 (Local History)
Let h be a history and O an object. The projection of h to O is the local history that results from removing all events from h which are not issued by O. The function $h\restriction_O$ is defined as follows:

$$(h \circ h')\restriction_O = h\restriction_O \circ h'\restriction_O$$

$$[\mathsf{invEv}(O', O'', f, m, \vec{e})]\restriction_O = \begin{cases} [\mathsf{invEv}(O', O'', f, m, \vec{e})] & \text{if } O' = O \\ \epsilon & \text{otherwise} \end{cases}$$

$$[\mathsf{invREv}(O', O'', f, m, \vec{e})]\restriction_O = \begin{cases} [\mathsf{invREv}(O', O'', f, m, \vec{e})] & \text{if } O' = O \\ \epsilon & \text{otherwise} \end{cases}$$

$$[\mathsf{futEv}(O', f, m, e)]\restriction_O = \begin{cases} [\mathsf{futEv}(O', f, m, e)]] & \text{if } O' = O \\ \epsilon & \text{otherwise} \end{cases}$$

$$[\mathsf{futREv}(O', f, e)]\restriction_O = \begin{cases} [\mathsf{futREv}(O', f, e)] & \text{if } O' = O \\ \epsilon & \text{otherwise} \end{cases}$$

$$[\mathsf{throwEv}(O', f, m, e)]\restriction_O = \begin{cases} [\mathsf{throwEv}(O', f, m, e)]] & \text{if } O' = O \\ \epsilon & \text{otherwise} \end{cases}$$

$$[\mathsf{throwREv}(O', f, e)]\restriction_O = \begin{cases} [\mathsf{throwREv}(O', f, e)] & \text{if } O' = O \\ \epsilon & \text{otherwise} \end{cases}$$

$$[\mathsf{awaitEv}(O', f, f')]\restriction_O = \begin{cases} [\mathsf{awaitEv}(O', f, f')] & \text{if } O' = O \\ \epsilon & \text{otherwise} \end{cases}$$

Definition 13 (System)
A system is a tuple $(\overrightarrow{Sys}, \overrightarrow{Sch}, h)$ where $\overrightarrow{Sys}$ is a partial function mapping $\mathbb{N}$ to objects, $\overrightarrow{Sch}$ is a function mapping objects to schedulers and h is a history.

We leave the set of schedulers underspecified, the ABS semantics are defined for any scheduler. We only aussume the following:

- The set of schedulers is not empty
- A scheduler sch is a partial function that maps the set $(\texttt{start} \times \mathsf{Met} \times \mathsf{Fut}) \cup (\texttt{react} \times \mathsf{Fut})$ to other schedulers. A function application $sch((\texttt{start}, m, f))$ queries whether a process

for the method m can be activated, an application $sch(\texttt{react}, f)$ queries whether the process computing the value of f can be reactivated. If a query succeeds, sch returns a new scheduler, otherwise $\perp$.

We write S for the set of all systems and $his(S)$ for the history of a system S. The semantics of systems is defined by the following 4 rules:

- The rule **internal** handles the case where an object o is reduced to o', without adding an event to the local history. The partial mapping $\overrightarrow{Sys}$ is updated to $\overrightarrow{Sys}'$ by setting $\overrightarrow{Sys}' = \overrightarrow{Sys}[i \mapsto o']$ for the $i \in \mathbf{dom}(\overrightarrow{Sys})$ such that $\overrightarrow{Sys}[i]$.

$$\frac{\begin{array}{c} i \in \mathbf{dom}(\overrightarrow{Sys}) \quad \overrightarrow{Sys}[i] = o \quad o \rightarrow_F o' \quad his(o') = his(o) \\ F = \{f \mid \exists i \leq |h|.\ h[i] = \mathsf{invEv}(O, O', f, m, e)\} \end{array}}{(\overrightarrow{Sys}, \overrightarrow{Sch}, h) \rightarrow (\overrightarrow{Sys}[i \mapsto o'], \overrightarrow{Sch}, h)} \ \textbf{internal}$$

- The rule **communicate** handles the case where an object o is reduced to o', but adds an event ev to the local history.

$$\frac{\begin{array}{c} i \in \mathbf{dom}(\overrightarrow{Sys}) \quad \overrightarrow{Sys}[i] = o \quad o \rightarrow_F o' \quad his(o') = his(o) \circ [ev] \\ F = \{f \mid \exists i \leq |h|.\ h[i] = \mathsf{invEv}(O, O', f, m, e)\} \end{array}}{(\overrightarrow{Sys}, \overrightarrow{Sch}, h) \rightarrow (\overrightarrow{Sys}[i \mapsto o'], \overrightarrow{Sch}, h \circ [ev])} \ \textbf{communicate}$$

- The rule **start** starts a new process in an object o, if o does not have any active process. This is the case if there is an invocation event $ev = \mathsf{invREv}(O', O, f, m, e)$ in the global history without a corresponding invocation reaction event and o is inactive. Then ev is added to the global history and the local history of o and a new process is generated from ev and set as active for o. The scheduler is queried with $(\texttt{start}, m, f)$ where f is the new future for the started process.

$$\frac{\begin{array}{c} \exists i \leq |h|.\ h[i] = \mathsf{invREv}(O', O, f, m, e) \\ \forall j \leq |h|.\ h[j] \neq \mathsf{invREv}(O', O, f, m, e) \\ active(o') = f \qquad \overrightarrow{Sch}(O)(\texttt{start}, m, f) \neq \perp \\ \overrightarrow{Sch}' = \overrightarrow{Sch}[O \mapsto \overrightarrow{Sch}(O)(\texttt{start}, m, f)] \\ i \in \mathbf{dom}(\overrightarrow{Sys}) \qquad \overrightarrow{Sys}[i] = (O, P, \sigma, \perp, h'') \\ o' = (O, P, \sigma, \mathfrak{P}(ev), h'' \circ [\mathsf{invREv}(O', O, f, m, e)] \end{array}}{(\overrightarrow{Sys}, \overrightarrow{Sch}, h) \rightarrow (\overrightarrow{Sys}[i \mapsto o'], \overrightarrow{Sch}', h \circ [\mathsf{invREv}(O', O, f, m, e)])} \ \textbf{start}$$

- The rule **continue** starts a process that was suspended before in an object o. The object o must be inactive and is transformed to the object o' by chosing a process from the process pool P which has **await** f as its first statement, checking that f has been resolved, removing this statement and setting it as the active process. The scheduler is

queried with $(\texttt{react}, f')$ where f is the future of the reactivated process.

$$\frac{\begin{array}{c} i \in \mathbf{dom}(\overrightarrow{Sys}) \qquad \overrightarrow{Sys}[i] = (O, P, \sigma, \bot, h') \\ o' = (O, P[j \mapsto \bot], \sigma, (\pi, \sigma', \rho, h'', f'), h') \\ \overrightarrow{Sch}(O)(\texttt{react}, f') \neq \bot \\ \overrightarrow{Sch}' = \overrightarrow{Sch}[O \mapsto \overrightarrow{Sch}(O)(\texttt{react}, f')] \\ j \in \mathbf{dom}(P) \quad P[j] = (\mathbf{await}\ f; \pi, \sigma', \rho, h'', f') \end{array}}{(\overrightarrow{Sys}, \overrightarrow{Sch}, h) \to (\overrightarrow{Sys}[i \mapsto o'], h)} \ \mathbf{continue}$$

A system S is terminated, written $\mathsf{terminated}(S)$ if all its objects are terminated. A system S is dead-locked, if it can not make any further execution step and not all of its objects are terminated. The big-step semantics describe when a system (object, process) realizes a history.

Definition 14 (Big-Step Semantics)
A system S generates a history h, if it can terminate in a state S_n with $his(S_n) = h$:

$$S \Downarrow h \iff \exists S_1, \dots, S_n \in \mathsf{S}.\ S \to S_1 \to \cdots \to S_n \wedge his(S_n) = h \wedge \neg\exists S' \in \mathsf{S}.\ S_n \to S'$$

Note that the scheduler is not purely functional: Upon (re-)activating a process, the scheduler may change. The scheduler may have a store and keep track of (re-)activated futures. Thus it is possible to define a scheduler which changes its behavior after counting or handling a certain method. We use a scheduler with a store for futures in Section 4.3. To specify how the values futures are read, we must be able to keep track of the propagation of futures. For simplicity, we only specify that if a future is communicated via a return value, then the return type of the corresponding value is an algebraic data type and all the future are available at the first level of the return value.

With these restrictions, we can represent a return value as the constructor and a list of pairs (i, v) with the meaning that the ith parameter is the value v.

Example 5
Let A be an algebraic data type with one constructor c, that has arity 3 and takes two integer and one future parameter.

```
A m(){ return c(10,100,f);}
```

The return value may be described by $(c, \{(1, 10), (2, 100), (3, f)\})$, where the pair $(1, 10)$ describes that the first parameter is the integer 10.

We use the description of parameters as their position in the parameter list to encode how futures are passed.

2.2. ABSDL

In this section we describe the ABS Dynamic Logic (ABSDL), which is used to specify class invariants. A class invariant is a description of an object's inner state and history that must hold upon object instantiation and whenever any process releases control or terminates. Such invariants are guarantees between processes.

To model the memory of objects, ABSDL uses *heap functions* which map locations to values. A location is a pair of an object and a field and a value is an element of some domain D. A *heap* is a function that maps locations to domain values. We write *Heap* for the set of all heaps and *Loc* for the set of all locations:

$$Heap = \mathsf{Ob} \times Fl \to D$$
$$Loc = \mathsf{Ob} \times Fl$$

To verify that a method with method body m satisfies an invariant φ, ABSDL checks the validity of $\varphi \Rightarrow [m]\varphi$: if φ holds before executing m, then it holds afterwards if m terminates. The verification tool KeY-ABS can only handle partial correctness, i.e. it assume that all loops are always executed finitely often.

ABSDL extends first-order logic with updates and symbolic execution. Updates have the form $v := t$ and denote that the value of the variable v is changed to t. As the object history and memory are handled as special program variables these updates can denote all state changes. Symbolic execution denotes that a modality $[s]$ of statements s is unrolled one statement at a time. The first statement inside the modality is symbolically executed by removing the statement and appending an update that encapsulates the effect of the statement.

Symbolic execution does not work on the actual values, but on symbolical ones which describe a possible set of values. For example, the formula $o.f > 0 \Rightarrow i = o.f * 2$ describes that the variable i contains a value which is the twice as big as the value of $o.f$, issuming that $o.f$ is strictly positive.

Definition 15 (Signature)
Let $(\mathsf{A}, \mathsf{C}, \mathsf{I})$ be an ABS data type system. A *signature* is a tuple $\Sigma = (\mathsf{F}, \mathsf{P}, \mathsf{V}, \mathsf{PV}, \mathsf{S})$ where F is a set of function symbols, P is a set of predicate symbols, V is a set of logical variables, PV is a set of program variables and S is the set of sorts.

We assume that every ABS class, interface and abstract data type is a sort and that the set of sorts is closed under future types, sequences and sets. Additionally we demand that types for heap memory, locations and fields are in S. *Any* is a special sort s.t. every sort is a subset of *Any*.

$$\mathsf{A} \subseteq \mathsf{S} \wedge \mathsf{C} \subseteq \mathsf{S} \wedge \mathsf{I} \subseteq \mathsf{S}$$
$$s \in \mathsf{S} \to \{Seq\langle s\rangle, Set\langle s\rangle, Fut\langle s\rangle\} \subset \mathsf{S}$$
$$\{Any, Heap, LocSet, Field\} \subseteq \mathsf{S}$$

We write a function or predicate symbol f with arity n as f_n and assume the following properties:

- All sets in the signature are pairwise disjoint.
- There is a program variable for the heap memory: $heap : Heap \in \mathsf{PV}$.
- There is a program variable for the history: $history : History \in \mathsf{PV}$.
- There is a function symbol fl^{Σ} for every field fl of every ABS class we reason about, and additionally the $created_0$ field with $created_0 : Field$.
- There is a function symbol C_0^{Σ} for every constructor C of an abstract data type.

- The following function symbols are used to model the heap, sets, sequences and events:
  - For the heap: $store_4 : Heap,\ select_3 : Any,\ anon_3 : Heap$
  - For sets: $singleton_1, union_2$
  - For sequences: $seqSingleton_1, seqEmpty_0, seqConcat_2, issuedBy_2$
  - For events: $\mathsf{invEv}_5^\Sigma, \mathsf{invREv}_5^\Sigma, \mathsf{futEv}_4^\Sigma, \mathsf{futREv}_3^\Sigma, \mathsf{throwEv}_4^\Sigma, \mathsf{throwREv}_3^\Sigma, \mathsf{awaitEv}_3^\Sigma$
- There are predicate symbols $wellFormed_1, pattern_3, outer_2, paramAt_4 \in \mathsf{P}$.

We mostly omit the arity index and the Σ superscript in the rest of this work for readability.

Definition 16 (Syntax)
Given a fixed signature Σ, we define formulas φ, terms t and updates U by the following grammar, where p ranges over P, f over F, T over S, v over PV, x over V, n over $\mathbb{Z}$ and π over the set of ABS statements:

$$\begin{aligned}
U &::= \ v := t \mid U || U \mid \{U\}U \\
\varphi &::= \ \mathbf{true} \mid \mathbf{false} \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid p(t \dots t) \mid t \geq t \mid t < t \mid \\
&\quad\ \ \forall T\ x; \varphi \mid \exists T\ x; \varphi \mid [\pi]\varphi \mid \{U\}\varphi \mid \varphi \leftrightarrow \varphi \mid \varphi \rightarrow \varphi \\
t &::= \ v \mid \{U\}t \mid f(t \dots t) \mid t \doteq t \mid n \mid t + t \mid t - t \mid \mathit{TRUE} \mid \mathit{FALSE}
\end{aligned}$$

We denote the set of all formulas with Frm and the set of all terms with Trm. We only consider well-typed formulas and terms with respect to the sorts. We again refrain from giving a formal type system for terms and formulas. The extensions needed for encoding session types are only additional predicate and function symbols. A formal treatment can be found e.g. in [30].

Example 6
Consider the following formula:

$$\{heap := store(heap, o, fl, 10)\}\exists Int\ i; \big(select(heap, o, fl) = 2 * i \wedge i > 0\big)$$

It expresses that in a state after the value 10 is stored in $o.fl$, the value of $o.fl$ is even.

Definition 17 (Interpretation and Configuration)
Let D be the domain, a non-empty set of values. Let $\mathcal{I}$ be an interpretation that maps every function symbol $f_n \in \mathsf{F}$ to a function $\mathcal{I}(f_n) : D^n \rightarrow D$ (where the elements of the preimage are typed according to the function) and every predicate symbol $p_n \in \mathsf{P}$ to a predicate $\mathcal{I}(p_n) \subseteq D^n$. A configuration is a function $\sigma : \mathsf{PV} \rightarrow D$ denoting the current values of the program variables.

The mandatory function symbols are interpreted as follows.

$$
\begin{aligned}
\mathcal{I}(C^{\Sigma}) &= C \\
\mathcal{I}(fl^{\Sigma}) &= fl \\
\mathcal{I}(select) &: Heap \times \mathsf{Ob} \times Fl \to D \\
\mathcal{I}(select)(heap, ob, fl) &= heap(ob, fl) \\
\mathcal{I}(store) &: Heap \times \mathsf{Ob} \times Fl \times D \to Heap \\
\mathcal{I}(store)(heap, ob, fl, d) &= heap[(ob, fl) \mapsto d] \\
\mathcal{I}(anon) &: Heap \times \mathcal{P}(\mathsf{Ob} \times Fl) \times Heap \to Heap \\
\mathcal{I}(anon)(heap_1, locset, heap_2) &= heap_1[(o, fl) \mapsto heap_2(o, fl)]_{(o,fl) \in locset} \\
\mathcal{I}(singleton) &\in \mathcal{P}(D) \\
\mathcal{I}(singleton)(d) &= \{d\} \\
\mathcal{I}(union) &\in \mathcal{P}(D) \times \mathcal{P}(D) \to \mathcal{P}(D) \\
\mathcal{I}(union)(A_1, A_2) &= A_1 \cup A_2 \\
\mathcal{I}(seqEmpty) &\in D^* \\
\mathcal{I}(seqEmpty) &= \epsilon \\
\mathcal{I}(seqSingleton) &\in \mathcal{P}(D) \\
\mathcal{I}(seqSingleton)(d) &= [d] \\
\mathcal{I}(seqConcat) &\in D^* \times D^* \to D^* \\
\mathcal{I}(seqConcat)(s_1, s_2) &= s_1 \circ s_2 \\
\mathcal{I}(pattern) &\subseteq \mathsf{Ev}^* \times \mathcal{P}(\mathsf{Ev}^*) \times \mathbb{N} \\
\mathcal{I}(pattern)(s, S, k) &\iff \exists s' \in \mathcal{L}(S^k).\ s \equiv_{\mathsf{Fut}} s' \\
\mathcal{I}(outer) &\subseteq D \times C \\
\mathcal{I}(outer)(d, c) &\iff d = c(\vec{e}) \text{ for some parameters } \vec{e} \\
\mathcal{I}(paramAt) &\subseteq D \times D \times \mathbb{N} \\
\mathcal{I}(paramAt)(d_1, d_2, i) &\iff \exists C \in \mathsf{Con}.\ d_1 = C(\vec{e}) \land e[i] = d_2 \text{ for some parameters } \vec{e} \\
\mathcal{I}(issuedBy) &\subseteq \mathsf{Ev}^* \times \mathsf{Fut}
\end{aligned}
$$

Intuitively, $\mathcal{I}(issuedBy)(S, f)$ holds if every event in S has been issued by the process computing f. The definition for invREv, awaitEv, futEv, and throwEv is straightforward. E.g., for awaitEv:

$$
\begin{aligned}
\mathcal{I}(\mathsf{awaitEv}^{\Sigma}) &: \mathsf{Ob} \times \mathsf{Fut} \times \mathsf{Fut} \to \mathsf{Ev} \\
\mathcal{I}(\mathsf{awaitEv}^{\Sigma})(O, f, f') &= \mathsf{awaitEv}(O, f, f')
\end{aligned}
$$

We could add additional parameters to invEv, throwREv and futREv to keep track of the active future or add auxiliary structures like a special program variable that maps positions in the history to the future which issued them. For presentations sake, as the main focus of this work is not extending ABS or ABSDL, we let $\mathcal{I}(issuedBy)$ underspecified.

For readability we write ϵ for *seqEmpty* and analogously for *seqSingleton* and *seqConcat*. Also we write $o.fl$ for $select(heap, o, fl)$.

The semantics of the modality is a *Kripke Structure* which models state transition with transitions between configurations.

Definition 18 (Kripke Structure)
A Kripke Structure is a tuple $(D, I, \mathsf{Conf}, \delta)$, where D is the domain, I the interpretation, Conf the set of configurations and the function $\delta : \mathsf{ABS} \to (\mathsf{Conf} \to \mathcal{P}(\mathsf{Conf}))$ models state transitions. A statement π is mapped on a function from configurations to configuration. This function models the effect of π on the memory. We write $\delta_\pi(\sigma)$ instead of $\delta(\pi)(\sigma)$.

The semantics is defined with a evaluation function which is dependent on a configuration and a assignment on logical variables.

Definition 19 (Semantics)
Let $\beta : \mathsf{V} \to D$ be an assignment on logical variables. We define the semantics of ABSDL formulas, terms and updates with a function $val_{\sigma,\beta}$ which maps formulas to truth values. terms to elements of the domain and updates to functions $\mathsf{Conf} \to \mathsf{Conf}$.

The semantic of terms and the FO-fragment of ABSDL is standard and we refrain from presenting it. The semantics of modality and updates are:

$$\begin{aligned}
val_{\sigma,\beta}(v := t)(\sigma') &= \sigma'[v \mapsto val_{\sigma,\beta}(t)] \\
val_{\sigma,\beta}(u_1 || u_2)(\sigma') &= val_{\sigma,\beta}(u_2)(val_{\sigma,\beta}(u_1)(\sigma')) \\
val_{\sigma,\beta}(\{u\}t) &= val_{val_{\sigma,\beta}(u)(\sigma),\beta}(t) \\
val_{\sigma,\beta}([\pi]\varphi) &= \begin{cases} \mathbf{tt} & \text{if } \forall \sigma' \in \delta_\pi(\sigma).\ val_{\sigma',\beta}(\varphi) = \mathbf{tt} \\ \mathbf{tt} & \text{if } \delta_\pi(\sigma) = \emptyset \\ \mathbf{ff} & \text{otherwise} \end{cases}
\end{aligned}$$

Definition 20 (Validity)
An ABSDL formula φ is valid iff $val_{\sigma,\beta} = \mathbf{tt}$ for all configurations σ and all variable assignments β.

Definition 21 (Invariant)
Let C be a class and φ a formula which only contains fields of this class and constants as terms.

The formula φ is an invariant for C, iff

$$\varphi \to [\mathbf{try}\{s\}\mathbf{catch}(\text{Exception } e)\{\mathbf{default} =>\}]\varphi$$

holds for every method-body s in C and φ holds at every process release point, i.e. after the execution of every **await** statement.

2.3. Calculus

A sound calculus for ABSDL has been developed in [15], which is based on the JavaDL calculus in [30]. We refrain from presenting the whole calculus and only present some rules to illustrate symbolic execution.

Definition 22 (Sequent)
A *sequent* is a pair of formula sets $(\Gamma, \Delta) \in \mathsf{Frm} \times \mathsf{Frm}$, written $\Gamma \Rightarrow \Delta$.
We write $\Gamma, \varphi_1 \dots \varphi_n \Rightarrow \psi_1 \dots \psi_n, \Delta$ for the sequent $\Gamma \cup_n \{\varphi_n\} \Rightarrow \Delta \cup_n \{\psi_n\}$
A sequent $\varphi_1 \dots \varphi_k \Rightarrow \psi_1 \dots \psi_l$ encodes the formula $\bigwedge_k \varphi_k \to \bigvee_l \psi_l$:

$$val_{\sigma,\beta}(\{\varphi_1 \dots \varphi_k\} \Rightarrow \{\psi_1 \dots \psi_l\}) = val_{\sigma,\beta}(\bigwedge_k \varphi_k \to \bigvee_l \psi_l)$$

Definition 23 (Rule)
A *rule* is a pair $(\mathsf{Seq}^*, \mathsf{Seq})$ where the first component is called *premisses* and the second one *conclusion*. A rule $(p_1 \dots p_n, q)$ is *sound* if for all configurations c the following property holds:

$$\bigwedge_{i \leq n} p_i \Rightarrow q$$

We use the following rule schema to denote a rule $(p_1 \dots p_n, q)$:

$$\frac{p_1 \qquad \dots \qquad p_n}{q}\ \text{name}$$

A rule without premisses is called *axiom*. We prove a sequent by building a proof tree.

Definition 24 (Proof Tree)
A proof tree is a tree where the nodes are sequents or the symbol **close**. The proof tree for a sequent S has S as its root. For each node q that is a sequent there is a rule, such that all children $(p_1 \dots p_n)$ are the premisses of this rule and q is the conclusion. If a rule has no premisses the node q has **close** as its sole child.

A proof for a sequent S is a tree where all leaves are **close**.
The formulas in a rule may contain schematic variables which have to be instantiated with terms or formulas, before the rule can be applied to a sequent.
If a rule has side conditions to the instantiations of the schematic variables we write them on the left side, while the name is written on the right side. We write down proof trees with the root sequent at the bottom, with children above their parent node and the rule denotations in between.

Definition 25 (Rewrite Rules)
A *rewrite rule* is a triple $(t_1, t_2, \varphi) \in (\mathsf{Trm} \times \mathsf{Trm} \times \mathsf{Frm}) \cup (\mathsf{Frm} \times \mathsf{Frm} \times \mathsf{Frm})$ written

$$t_1 \rightsquigarrow t_2 \ \textbf{if}\ \varphi$$

with the meaning that in every sequent we can always replace t_1 with t_2 in any formula if the side condition φ holds. We denote each rewrite rule with a name and omit the side condition if the side condition is *true*.

A modality is resolved by symbolic execution rules, which encapsulate the effect of the first statement in an update. As an example, the following rule is used for asynchronous method calls without parameters on objects store in fields:

$$\frac{\begin{array}{c}\Gamma \Rightarrow \{U\}(\mathit{select}(\mathit{heap}, \mathit{self}, \mathit{fl}) \neq \mathbf{null}), \Delta \\ \Gamma \Rightarrow \big\{U\big\}\big\{v := f || \mathit{history} = \mathit{history} \circ [\mathsf{invEv}(\mathit{self}, \mathit{self}.\mathit{fl}, f, m, \epsilon)]\big\}[\pi]\varphi, \Delta\end{array}}{\Gamma \Rightarrow \{U\}[\mathit{Fut}\langle T\rangle\ v\ =\ \mathit{fl}!m(); \pi]\varphi, \Delta}\ \text{Call} \quad f \text{ is fresh}$$

Try blocks and exceptions are handled as in Java, except that the catch block is transformed into a `case` statement instead of directly choosing the correct one. The rules can be found e.g. in [30].

The *paramAt* and *outer* predicates can be unrolled to a first-order formula. For each constructor C with arity n, we define two rules, both with the side-condition that i is a numeral smaller n.

$$\mathit{paramAt}(t_1, t_2, i) \rightsquigarrow \exists \mathsf{Con} C.\ \exists \mathit{Any}\ e_1, \ldots, e_n;\ t_1 = C(e_1, \ldots, e_n) \land e_i \doteq t_2$$

$$\mathit{outer}(t_1, C) \rightsquigarrow \exists \mathit{Any}\ e_1, \ldots, e_n;\ t_1 = C(e_1, \ldots, e_n)$$

The $\mathit{paramAt}(t_1, t_2, i)$ predicate describes that the term t_1 has C as its outermost constructor and the ith parameter is t_2. The $\mathit{outer}(t_1, C)$ predicate describes that the term t_1 has C as its outermost constructor.

The following rule splits a *pattern* expression into three cases, depending how often the set of sequences is repeated.

$$\begin{array}{rl} & s \doteq \epsilon \land k \doteq 0 \\ \mathit{pattern}(s, S, k) \rightsquigarrow & \lor \exists \mathit{Seq}\langle\mathsf{Ev}\rangle s';\ (s' \in S \land s \doteqdot_{\mathsf{Fut}} s') \land k \doteq 1 \\ & \lor \exists \mathit{Seq}\langle\mathsf{Ev}\rangle s', s'';\ s = s' \circ s''' \land \mathit{pattern}(s', S, k-1) \land \mathit{pattern}(s'', S, 1) \land k > 1 \end{array}$$

To check future equality we introduce a static check. This is needed because substitution is only supported for program variables with updates, but we need to check arbitrary terms.

Definition 26
Let $s_1, s_2 \in \mathsf{Trm}$ be two closed terms, i.e. they contain no free variables, and without updates. We say that s_1 and s_2 are *syntactically future equivalent*, if we can obtain s_2 from s_1 be replacing *function symbols* of future-type and arity 0 consistently Let $\{f_1, \ldots, f_n\}$ be the set of all such function symbols occurring in s_2. Then

$$\exists f'_1, \ldots, f'_n \in \mathsf{Fut}.\ s = s'[f_1 \backslash f'_1] \ldots [f_n \backslash f'_n]$$

Where $[f \backslash f']$ is syntactical substitution of every occurrence of f by f', must hold.

Note that this is a *check*, we do *not* substitute the symbols. This check allows us to formulate a simple rule for $\doteqdot_{\mathsf{Fut}}$

$$s_1 \doteqdot_{\mathsf{Fut}} s_2 \rightsquigarrow \mathbf{true\ if}\ s_1 \text{ and } s_2 \text{ are syntactically future equivalent}$$

To symbolically execute a loop one can either unroll it, i.e. transform it into a branching, or provide a loop invariant. A loop invariant is a formula that describes the states before the execution of the loop and and the execution of each iteration. This allows to approximate the post state after executing the whole loop.

In [30] a loop invariant rule for JavaDL was introduced, which uses *anonymizing*. Anonymizing marks a part of the heap memory as unchanged, i.e. it is not written during a loop iteration. This allows to keep information from the pre-state without directly encoding it into the loop invariant. We use a variant of this rule. The rule for ABSDL is more simple because ABS does not have break and continue statements. The subcase presented here also has no return statement inside the loop body.

$$\frac{\begin{array}{l}\Gamma \Rightarrow \{U\}inv, \Delta \\ \Gamma, \{U\}\{U_A\}(inv \wedge g) \Rightarrow \{U\}\{U_A\}[bd]inv, \Delta \\ \Gamma, \{U\}\{U_A\}(inv \wedge \neg g) \Rightarrow \{U\}\{U_A\}[\pi]\varphi, \Delta\end{array}}{\Gamma \Rightarrow \{U\}[\mathbf{while}(g)bd;\ \pi]\varphi, \Delta} \ \textbf{loopInvariantAnon}$$

Where

$$\begin{aligned} U_A = \{&history := history \circ s \ || \ old_history := history \ || \\ &heap := anon(heap, T, heap') \ || \ v_1 := v_1' \ || \ \ldots \ || \ v_m := v_m'\} \end{aligned}$$

for a given set of locations T, formula φ, all local variables $v_1, \ldots, v_m$ which occur on the left-hand side of assignment in bd and fresh function symbols $s, heap', v_1', \ldots, v_m'$ of fitting sorts.

The first branch shows that the invariant holds in the pre-state. The second branch shows that the invariant is preserved by the loop body. The third branch continues the proof.

Example 7
Consider the sequent

$$i > 0 \Rightarrow \big[\mathbf{while}(i > 0)\{Fut\langle Int\rangle f = fl!m();\}\big]i \doteq 0$$

A fitting loop invariant would be

$$inv = \exists Seq\langle \mathsf{Ev}\rangle s';\ history \doteq old_history \circ s' \wedge \exists Fut f';\ \exists Int\ k.\ pattern(s', \{[\mathsf{invEv}(self, self.f, m, f')]\}, k)$$

$$\begin{aligned} inv = \ & i \geq 0 \ \wedge \\ & \exists Seq\langle \mathsf{Ev}\rangle s';\ history \doteq old_history \circ s' \wedge \\ & \exists Fut f';\ \exists Int\ k.\ pattern(s', \{[\mathsf{invEv}(self, self.f, m, f')]\}, k) \end{aligned}$$

After applying the **loopInvariantAnon** rule there are three open branches

- The first branch has the form

  $$i > 0 \Rightarrow inv$$

  This case can be closed by choosing $s' = \epsilon$ and $k = 0$

- The second branch has (simplified) the form

  $$i > 0, pattern(s'', \{[\mathsf{invEv}(self, self.f, m, f'], k') \Rightarrow \{history = s''\}[Fut\langle Int\rangle f = self.f.m()!;]inv$$

  After executing the method body it has the form

  $$i > 0, pattern(s'', \{[\mathsf{invEv}(self, self.f, m, f'], k') \Rightarrow \{history = s'' \circ [\mathsf{invEv}(self, self.f, m, f'']\}inv$$

Thus it remains to show

$$\exists Seq\langle \mathsf{Ev}\rangle s';\ history \doteq s'' \circ s' \land \exists Fut f';\ \exists Int\ k;\ pattern(s', \{[\mathsf{invEv}(self, self.f, m, f']\}, k)$$

under the premise

$$pattern(s'', \{[\mathsf{invEv}(self, self.f, m, f'], k'))$$

This can be done by instantiating s' with $[\mathsf{invEv}(self, self.f, m, f'', \epsilon]$, k with k', f' with f'' and showing that

$$[\mathsf{invEv}(self, self.f, m, f'', \epsilon] \doteq_{\mathsf{Fut}} \mathsf{invEv}(self, self.f, m, f'', \epsilon]$$

This holds as both sides of the equation are syntactically equivalent.

- The last branch has (simplified) the form

$$i \geq 0, i \leq 0 \Rightarrow i \doteq 0$$

This also obviously holds.

3. Session Types

Session types specify a communication between endpoints. These communications can be viewed as the history a system realizes. Thus session types specify the set of valid *histories*. A system is captured by a type if all histories it can realize are described by the session type. Session types are an established specification language for channel-based asynchronous communication [29, 7, 18]. With channels, three actions must be modeled at global level: Sending data, choosing a communication and repeating a communication.

The ABS concurrency models requires more actions. The endpoints are now processes, which are bundled by their object. To specify cooperative scheduling, our session types have actions for releasing and regaining control. Sending data can be done in two ways: by a method call or by returning a value and reading it. We handle the method call analogously to sending data over a channel in e.g. [29] but use different actions for resolving a future and reading from it, than for sending and receiving data via a method call. The reason is that several processes can read from the same future, there may be arbitrary actions between resolving and reading and that a process may read several times from the same future.

These additional actions, the cooperative scheduling and the fixed order of operations on futures lead to a model with a more complex concept of *well-formedness of types* compared to the model for asynchronous communication on channels in [18]. The complexity arises because at every event one must check the already executed communication events. E.g. it must be ensured that when a future is resolved at position i, it must have been used to start a process before and that this process is currently active in its object.

The different ways to send data also require a more convoluted handling of branching. Channel-based systems can label each branch with a label and send the label to communicate which branch has been chosen to resume the session. In the ABS concurrency model each communication either starts or terminates a process and can not be used for arbitrary data:

- If a process needs to communicate a choice to an already running process, this must be encoded into the return value if further data has to be passed.
- If a process needs to communicate a choice to a not running process, this must be encoded into the call.

We denote the first case as *backward choice* and the second as *forward choice*. In one branching backward and forward choices may be mixed, but backward choices require additional concepts, because the choice is read by the receiving process when accessing the communicating future, not when the future is resolved.

We follow mostly [18] for syntax and presentation. In this Chapter we present the syntax and semantics of types, the projection of global to local type, the translation into an ABSDL invariant and describe under which conditions we can state a fidelity theorem.

3.1. Global Types

A global type describes the communication within a closed system of objects.

To reduce the complexity introduced by asynchronous calls and delays, the global type describes *one* possible run of the system. In this run, everything happens exactly in the same order as specified and without delays. This allows a compact representation of concurrent systems.

To reason about the actual systems with asynchronous calls and delays, the system may not only realize the history of this run, but a *permutation* of it. But the reorderings of events must be invisible to any object. Every object can not distinguish the simple system without delays and the real system with delays.

In this section we present the syntax of global types. The well-formedness conditions are presented in Section 3.5, the translation into a regular expression in Section 3.4.

To initiate the communication we use a special object $\mathbf{0}$. The first action in every type must be a call from $\mathbf{0}$ to some other object. Afterwards $\mathbf{0}$ is not part of the communication, i.e. it can not be called and has no active processes.

Definition 27 (Global Types)
Let J range over $\mathcal{P}(\mathbb{N})$ with $|J| \geq 1$, $\mathbf{p}, \mathbf{q}$ over Ob, f over Fut, m over Met, C over Con, $R \in \mathcal{P}(\mathsf{Fut} \times \mathbb{N})$ over sets of pairs of futures and numbers.

The syntax of global types is defined by

$$\begin{aligned}\mathbf{G} \quad ::= \quad & \mathbf{p} \xrightarrow{f} \mathbf{q}\!:\!m(R) \;.\; \mathbf{G} \;\big|\; \mathbf{p}{\downarrow}f\!:\!(C,R) \;.\; \mathbf{G} \;\big|\; \mathbf{p}{\uparrow}f\!:\!(C,R) \;.\; \mathbf{G} \;\big| \\ & \mathsf{Rel}(\mathbf{p},f) \;.\; \mathbf{G} \;\big|\; \mathbf{p}\{\mathbf{G}_j\}_{j\in J} \;\big|\; \mathbf{G}^* \;.\; \mathbf{G} \;\big|\; \mathsf{end}\end{aligned}$$

If a type does not have the form $\mathsf{end}, \mathbf{G} \;.\; \mathbf{G}', \mathbf{p}\{\mathbf{G}_j\}_{j\in J}$ or $\mathbf{G}^*$ we say the type is *simple*.

The call action $\mathbf{p} \xrightarrow{f} \mathbf{q}\!:\!m(R)$ models that the currently active process at the object $\mathbf{p}$ calls method m on object $\mathbf{q}$ and $\mathbf{q}$ starts a new process. The call is handled by the future f and other futures are passed as described by R. E.g. for $R = \{(i, f_1), (j, f_2), \dots\}$ the future f_1 is passed as the ith parameter and the future f_2 as the jth parameter, etc. A call action corresponds to an asynchronous call in ABS and to a sequence of an invocation and an invocation reaction event in histories. If no futures are passed we omit the R parameter.

The resolve action $\mathbf{p}{\downarrow}f\!:\!(C,R)$ models that the object $\mathbf{p}$ finishes the computation of f and resolves the future. If the method has an algebraic data type as its return type, the return value has C as its outermost constructor and R describes how the futures are passed in the return value. This corresponds to a return statement in ABS or a resolving event in histories. If the return type is not an algebraic data type we write $\mathbf{p}{\downarrow}f\!:\!(\bot,\emptyset)$. In examples we omit C and R if $C = \bot$ or $R = \emptyset$ for readability.

The fetch action $\mathbf{p}\uparrow f : (C,R)$ models that the object $\mathbf{p}$ reads from the future f. If f is resolved by a value with an algebraic data type, the value has C as its outermost constructor and $\mathbf{p}$ reads the futures as described by R. If f is resolved by a value which is not typed with an algebraic data type we write $\mathbf{p}{\uparrow}f$. This corresponds to a get expression in ABS or a fetch event in histories. We omit the C and R parameters as described for the resolving action.

The release action $\mathsf{Rel}(\mathbf{p},f)$ models that object $\mathbf{p}$ releases control until the future f has been resolved. This corresponds to an **await** f; statement in ABS or an $\mathsf{awaitEv}$ event in histories.

Branching is denoted by $\mathbf{p}\{\mathbf{G}_j\}_{j\in J}$, repetition by $\mathbf{G}^*$ with the meaning that $\mathbf{G}$ is repeated finitly often, concatenation by $\mathbf{G}_1 \;.\; \mathbf{G}_2$ and termination by end.

Repetition does not repeat the exact same type, as this would not be a valid use of futures.

$$\mathbf{G}^* \neq \mathbf{G} \ . \ \mathbf{G} \ . \ \ldots$$

If the repeated type contains a call, the future would be reused in the second iteration. Thus repetition assumes that every iteration renames all futures consistently: all iterations must be future-equivalent.

Example 8
Consider a protocol P with objects $\mathbf{A}, \mathbf{B}, \mathbf{C}$, where $\mathbf{A}$ has a list of items and wants to register them at $\mathbf{B}$. To do so, $\mathbf{A}$ repeatedly calls the method *reg* on $\mathbf{B}$. The return type of *reg* is an algebraic data type with two constructors deny, ok, both without parameters. $\mathbf{B}$ may decide to deny a registration and returns deny. In this case $\mathbf{A}$ calls $\mathbf{C}$ and logs the failure. Otherwise, $\mathbf{B}$ asks $\mathbf{A}$ for additional information on the passed item by calling *more* and returns ok. After all items have been processed, $\mathbf{A}$ terminates the protocol.
The global session to describe this communication is:

$$\mathbf{0} \xrightarrow{f_0} \mathbf{A}\!:\!\mathit{start}. \left(\mathbf{A} \xrightarrow{f} \mathbf{B}\!:\!\mathit{reg} \ . \ \mathsf{Rel}(\mathbf{A}, f_0) \ . \ \mathbf{B} \begin{Bmatrix} \mathbf{B} \xrightarrow{f'} \mathbf{A}\!:\!\mathit{more} \ . \ \mathbf{A}{\downarrow}f' \ . \ \mathbf{B}{\uparrow}f' \ . \ \mathbf{B}{\downarrow}f\!:\!\mathsf{ok} \ . \ \mathbf{A}{\uparrow}f \\ \mathbf{B}{\downarrow}f\!:\!\mathsf{deny} \ . \ \mathbf{A}{\uparrow}f \ . \ \mathbf{A} \xrightarrow{f''} \mathbf{C}\!:\!\mathit{log} \ . \ \mathbf{C}{\downarrow}f'' \ . \ \mathbf{A}{\uparrow}f'' \end{Bmatrix}\right)^{*} . \mathbf{A}{\downarrow}f_0 . \mathsf{end}$$

This example also illustrates the restriction for the communication of choice. The object $\mathbf{A}$ is notified twice about the choice of $\mathbf{B}$: First by receiving a call on *more* or not, then by the return value in f'. The object $\mathbf{C}$ is notified once, by receiving a call from $\mathbf{A}$ or not. All these propagations are correct because every *process* is notified once and behaves the same up to the moment it receives the notification.

To define projection and semantics of global types we need the following auxiliary structures and functions. When checking an action, we must be able to look into the prefix of it to check whether it is using its future correctly. E.g., to check a resolving action $\mathbf{A}{\downarrow}f$ there must be a calling action before. To identify an action we enumerate the AST and identify an action with its number. The prefix of an action is the set of all actions which are in the path from the root to the action, or on a branch underneath this path as the left child of a concatenation but not under a node resulting from a repetition. The actions inside a repetition are omitted because a repetition can be repeated zero times. Also well-formed types, which are defined in Section 3.5, ensure that omitting the repetition does not break well-formedness of the described histories.

Example 9
Consider the following example:

$$\mathbf{G} = \mathbf{0} \xrightarrow{f_0} \mathbf{A}\!:\!\mathit{start} \ . \ \left(\mathbf{A} \begin{Bmatrix} \mathbf{A} \xrightarrow{f'} \mathbf{B}\!:\!m \ . \ \mathbf{B}{\downarrow}f' \ . \ \mathbf{A}{\uparrow}f' \\ \mathbf{A} \xrightarrow{f''} \mathbf{C}\!:\!m \ . \ \mathbf{C}{\downarrow}f'' \ . \ \mathbf{A}{\uparrow}f'' \end{Bmatrix}\right)^{*} \ . \ \mathbf{A}{\downarrow}f_0 \ . \ \mathsf{end}$$

Note that $\mathbf{A}$ never releases control and in the whole communication only one process of $\mathbf{A}$ is active. The AST of $\mathbf{G}$ is depicted in Figure 3.1
The node labelled 2 refers to the action $\mathbf{A}{\uparrow}f''$ and the grayed out nodes are its strict prefix. The node labelled 1 refers to the action $\mathbf{C}{\downarrow}f''$ and is the last action before $\mathbf{A}{\uparrow}f''$.
Let i be the number of the node 2, then the prefix type is:

$$\mathbf{0} \xrightarrow{f_0} \mathbf{A}\!:\!\mathit{start} \ . \ \mathbf{A} \xrightarrow{f''} \mathbf{C}\!:\!m \ . \ \mathbf{C}{\downarrow}f''$$

Note that while the nodes labelled with $\mathbf{A}$ and $*$ are part of the strict prefix in the AST, they are removed in the prefixtype.

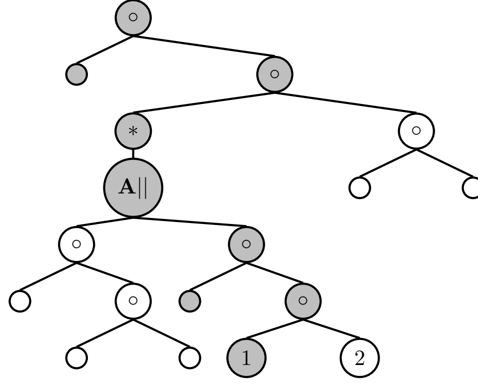


Figure 3.1.: Abstract syntax tree of $\mathbf{G}$

Formally we define the AST as follows:

Definition 28 (Abstract Syntax Tree)
Let $\mathbf{G}$ be a global type and $\mathfrak{T}(\mathbf{G}) = (V, E, N_\circ, N_*, (N_{\mathbf{p}||})_{\mathbf{p}\in\mathsf{Ob}})$ its AST, where V are the nodes corresponding to actions. An edge $(v, w) \in E$ describes that v corresponds to a type which is a subterm of the type of q. As the graph is a tree and the root is known, we regard E as undirected. The predicate $N_*(v)$ holds for nodes resulting from a $*$ operator, $N_\circ(v)$ holds for nodes resulting from a concatenation operator and $N_{\mathbf{p}||}(v)$ holds for nodes resulting from a global choice operator with $\mathbf{p}$ as the choosing object.

We identify V with the linear order of natural numbers produced by a DFS and denote with $\mathsf{action}(i)$ the ith visited node. We also write $child(i, j)$ for the number of the jth child of the node i, $parent(i)$ for the parent of i and $subs(i)$ for the set of all nodes j below i, such that there is a non-empty path from i to j that does not contain a node which is labelled with N_*.

Definition 29 (Prefix Set)
The $pre : \mathbb{N} \to \mathcal{P}(\mathbb{N})$ function is used to compute the prefix set of a leaf, i.e. the set of nodes, which denote actions that must be executed before i.

$$pre(i) = \begin{cases} \{i\} \cup pre(parent(i)) \cup subs(i+1) & \text{if } N_\circ(i) \wedge i \neq 0 \\ \{i\} \cup pre(parent(i)) & \text{if } \neg N_\circ(i) \wedge i \neq 0 \\ \{i\} & \text{if } i = 0 \end{cases}$$

The first case is the the parent of i is a node labelled with $\circ$. In this case the left subtree under $parent(i)$ must also be added, as the leaf-nodes in this subtree refer to actions which must be executed before i (or are under a node labelled with $*$ and are thus ignored by $subs$). In the first and the second case the parent node and its prefix are also added to the prefix. The last case is needed to add the root.

The *strict prefix* is defined as $spre(i) = pre(i) \setminus \{i\}$. The $last : \mathcal{P}(\mathbb{N}) \to \mathbb{N}$ function returns the last simple type in a set of vertices

$$last(I) = \max \left\{ i \in I \mid \neg(N_\circ(i) \vee N_{p||}(i) \vee N_*(i)) \right\}$$

The $last(I)$ function denotes the last action that must be executed before in I. If I is the prefix set of a node i, then $last$ denotes the last action which must be executed before i. $last$

always returns the last action which must be executed *in every case*, because in the prefix of a node there is no branching. Branching is not allowed on the left side of a concatenation, and thus would occur under a repetition node, and these are ignored when computing the prefix.

Analogous to the prefix set, we can define the prefix of a type at position i as the type which has only the nodes $pre(i) \setminus \{i\}$ as its AST.

Definition 30 (Prefix Type)
Let $\mathbf{G}$ be a type, $\mathfrak{T}(\mathbf{G})$ its AST with universe V and $i \in V$ a leaf. Let Q be the set of all non-repetitions nodes in the prefix of $\mathsf{action}(i)$:

$$Q = \{j \in pre(i) \mid \neg N_*(j)\}$$

The prefix type of $\mathbf{G}$ at i, $preType(\mathbf{G}, i)$ is the type such that the AST of $preType(\mathbf{G}, i)$ is the restriction of the AST of $\mathbf{G}$ on $pre(i)$:

$$\mathfrak{T}(pre(\mathbf{G}, i)) = (pre(i) \setminus Q, E', N'_\circ, N'_*, (N'_{\mathbf{p}||})_{\mathbf{p}\in\mathsf{Ob}})$$

and all relations C' agree with their counterpart in $\mathfrak{T}(\mathbf{G})$ on $pre(i)$ and the edge relation bypasses all repetitions:

$$E' = \{(i,j) \mid \neg N_*(i) \wedge N_*(j) \wedge \big(E(i,j) \vee \exists k.\ E(i,k) \wedge E(k,j)\big)\}$$

The strict prefix type $spreType(\mathbf{G}, i)$ is defined analogously over $spre(i)$ instead of $pre(i)$.

The prefix type of i is the the type that describes the communication that must be executed in every case before executing the action at i.

To keep track of active futures, we define a function which returns for each position i in the AST and each object $\mathbf{p}$ the active future $act(i, \mathbf{p})$. Every action in the global types corresponds to actions of the ABS system in the simplified model without delays. E.g. after $\mathbf{A} \xrightarrow{f} \mathbf{B}$, the object $\mathbf{B}$ is active and the active future is f, after $\mathbf{B}\downarrow f$ the object $\mathbf{B}$ is not active, etc. The active future of $\mathbf{p}$ at i is the future whose computing process is active, before the action of the ith node was executed. If there was no active future, $act(i, \mathbf{p})$ is undefined. Similarly we keep track which future is suspended, by mapping a future f to a set of pairs $(\mathbf{p}, f')$ to model that the computation of the future f' in object $\mathbf{p}$ is suspended until f is resolved.

Definition 31 (Active and Waiting Futures)
We define the set of active and waiting futures before executing the ith node in a fixed AST with the following two functions

$$act : \mathbb{N} \times \mathsf{Ob} \rightharpoonup \mathsf{Fut}$$
$$wait : \mathbb{N} \times \mathsf{Fut} \to \mathcal{P}(\mathsf{Ob} \times \mathsf{Fut})$$
$$act(0, \mathbf{p}) = \bot$$
$$\underset{i>0}{act(i, \mathbf{p})} = \begin{cases} \bot & \text{if } \exists f \in \mathsf{Fut}.\ last(spre(i)) = \mathsf{Rel}(\mathbf{p}, f) \\ \bot & \text{if } \exists f \in \mathsf{Fut}.\ last(spre(i)) = \mathbf{p}\downarrow f \\ f & \text{if } \exists \mathbf{q} \in \mathsf{Ob}.\ \exists f \in \mathsf{Fut}.\ last(spre(i)) = \mathbf{q} \xrightarrow{f} \mathbf{p} \\ f & \text{if } \exists \mathbf{q} \in \mathsf{Ob}.\ \exists f \in \mathsf{Fut}.\ last(spre(i)) = \mathbf{q}\downarrow f' \wedge wait(i, f') = \{(\mathbf{p}, f)\} \\ act(last(spre(i)), \mathbf{p}) & \text{otherwise} \end{cases}$$
$$wait(0, f) = \emptyset$$

$$\underset{i>0}{wait(i,f)} = \begin{cases} wait(last(spre(i)))(f) \cup \{(\mathbf{p}, act(last(spre(i)), \mathbf{p}))\} & \text{if } last(spre(i)) = \mathsf{Rel}(\mathbf{p}, f) \\ \emptyset & \text{if } \exists \mathbf{p} \in \mathsf{Ob}.\ last(spre(i)) = \mathbf{p}{\downarrow}f \\ wait(last(spre(i)))(f) & \text{otherwise} \end{cases}$$

The first case for the definition of *act* removes the currently active future if the last action was a release action. The second case removes the currently active future if the last action was a resolving action. The third case sets the currently active future to f, if the last action was a call on f. The forth case sets the currently active future to f, if the last action was a termination of f' and f waited for f'. This also demands that there is only one reactivated future. If there are several reactivated futures, the analysis is wrong for all following actions. However, we ensure that *act* is only used when there is only one reactivated future. The last case propagates the currently active future.

The first case in the definition of *wait* adds a pair, if an object releases control. The second case removes all pairs, once it is resolved and the third case propagates otherwise.

Additionally we define $fresh \subseteq \mathsf{Fut} \times \mathbb{N}$ as a predicate to check whether a future is fresh. In a given AST, $fresh(f,i)$ holds iff in the prefix of i the future f never occurs:

$$fresh(f,i) = \forall j \in spre(i).\ \forall \mathbf{p}, \mathbf{q} \in \mathsf{Ob}.\ \mathsf{action}(j) \neq \mathbf{p} \xrightarrow{f} \mathbf{q}$$

The *reads* function returns those objects which read from a given future in a given global type:

$$reads(\mathbf{G}, f) = \{\mathbf{p} \in \mathsf{Ob} \mid \mathbf{p}{\uparrow}f \text{ is a subterm of } \mathbf{G} \vee \mathbf{q} \xrightarrow{f} \mathbf{p} \text{ is a subterm of } \mathbf{G}\}$$

We do not provide a type system. Instead we regard global types as a proposition about the histories which a system can produce. Thus we can translate a global type into a regular expression. All histories which are described by the regular expression, have the events that correspond to the single actions in the same order as in the global type. I.e. the translated regular expression ignores concurrency and delays in the network and assumes a synchronous communication pattern which is coordinated according to the global type.

The formal translation of a global type into a regular expression is given in Section 3.4.

3.2. Local Types

Local types describe the communication within a closed system from the point of view of a single endpoint. We differ between *Object-Local Types* which describe all communication events executed by an object and *Method-Local Types* which describe all communication events executed by a single process. Both share the same syntax.

The distinction is necessary, because a single process does not have the information how it is interleaved with other processes. This information is only available at object-level - object-local types describe how method-local types relate to each other.

Definition 32 (Local Types)
Let J range over $\mathcal{P}(\mathbb{N})$ with $|J| > 1$, $\mathbf{p}$ over Ob, f over Fut, m over method names, C over constructor names and R over sets of pairs of futures and numbers. The syntax of local types is defined by

$$\begin{aligned} \mathbf{L} \ ::= \ & \mathsf{Put}\ f\!:\!(C,R)\ .\ \mathbf{L} \mid \mathsf{Get}\ f\!:\!(C,R)\ .\ \mathbf{L} \mid \mathbf{p}!_f m(R)\ .\ \mathbf{L} \mid \mathbf{p}?_f m(R)\ .\ \mathbf{L} \mid \\ & \mathsf{Await}(f,f')\ .\ \mathbf{L} \mid \mathsf{React}(f)\ .\ \mathbf{L} \mid \oplus\{\mathbf{L}_j\}_{j\in J} \mid \&_f\{\mathbf{L}_j\}_{j\in J} \mid \mathbf{L}^*\ .\ \mathbf{L} \mid \mathsf{skip}\ .\ \mathbf{L} \mid \mathsf{end} \end{aligned}$$

The resolve action $\mathsf{Put}\ f:(C,R)$ and the fetching action $\mathsf{Get}\ f:(C,R)$ have the same intuitive meaning as their global counterparts.

The send action $\mathbf{p}!_f m(R)$ denotes an asynchronous method call on the method m at the object $\mathbf{p}$. This corresponds to an invocation event in communication histories.

The receive action $\mathbf{p}?_f m(R)$ denotes the start of a new process, which computes f by executing the method m after a call from $\mathbf{p}$. This corresponds to an invocation reaction event.

The suspend action $\mathsf{Await}(f,f')$ denotes that the process computing f suspends its execution until the future f' has been resolved. The reactivation action $\mathsf{React}(f)$ denotes that the process computing f continues its execution. This action is needed, because the reactivation is implicitly encoded by the resolving action of the future which act as the guard. As local types only describe the view of a single object, this resolving action may not be visible and the effect of reactivation must be added.

The choice action $\oplus$ denotes that the currently active process actively chooses a branch to continue the execution of the protocol. The offer action $\&_f$ denotes that the object or process reacts on the choice of the process computing f and chooses a branch according to this choice. In case of a forward choice the currently active process behaves the same in every branch but new processes may be spawned (if called from f). In case of a backward choice the currently active process behaves the same in every branch up to the time it reads from f. Afterwards the branches may differ.

A method-local type describes the execution of a single process, thus it contains only one receiving action, only one resolving action and no end.

Definition 33 (Method-Local Types)
A local type is *method-local* for a future f iff

- its first action is $\mathbf{p}?_f m(R)$ for some $\mathbf{p}$, m, R,
- in every branch the last action is $\mathsf{Put}\ f:(C,R)$ for some C,R,
- it contains no further resolve action or receive action,
- and it contains no end.

If a local type is not method-local, it is *object-local*.

Method-Local types describe the behavior of a single object. Even if an object-local type only describes one future, it is not the method-local types of this future. First, the termination of the session end is not inside any process. Secondly, inside of repetition a future does not describe one process, but describes *multiple*: one process for every call. Each such process has the same communication pattern.

Example 10
Consider the protocol described in Example 8.

$$\mathbf{0} \xrightarrow{f_0} \mathbf{A}: start. \left(\mathbf{A} \xrightarrow{f} \mathbf{B}: reg\ .\ \mathsf{Rel}(\mathbf{A}, f_0)\ .\ \mathbf{B} \left\{ \begin{array}{l} \mathbf{B} \xrightarrow{f'} \mathbf{A}: more\ .\ \mathbf{A}\downarrow f'\ .\ \mathbf{B}\uparrow f'\ .\ \mathbf{B}\downarrow f:\mathsf{ok}\ .\ \mathbf{A}\uparrow f \\ \mathbf{B}\downarrow f:\mathsf{deny}\ .\ \mathbf{A}\uparrow f\ .\ \mathbf{A} \xrightarrow{f''} \mathbf{C}: log\ .\ \mathbf{C}\downarrow f''\ .\ \mathbf{A}\uparrow f'' \end{array} \right\}\right)^* . \mathbf{A}\downarrow f_0 . \mathsf{end}$$

The object-local type describing the behavior of $\mathbf{A}$ is

$$\mathbf{0}?_{f_0} start\ .\ \left(\mathbf{B}!_f reg\ .\ \mathsf{Await}\ (f_0, f)\ .\ \&_f \left\{ \begin{array}{l} \mathbf{B}?_{f'} more\ .\ \mathsf{Put}\ f'\ .\ \mathsf{React}\ f_0\ .\ \mathsf{Get}\ f:\mathsf{ok} \\ \mathsf{React}\ f_0\ .\ \mathsf{Get}\ f:\mathsf{deny}\ .\ \mathbf{C}!_{f''} log\ .\ \mathsf{Get}\ f'' \end{array} \right\}\right)^*\ .\ \mathsf{Put}\ f_0.$$

The method-local type of f' is

$$\mathbf{B}?_{f'}\mathit{more}\ .\ \mathsf{Put}\ f'$$

We use similar operations on the AST to work with local types as with global types.

Definition 34 (Abstract Syntax Tree for Local Types)
Let $\mathbf{L}$ be a local type and $\mathfrak{T}(\mathbf{L}) = (V, E, N_\circ, N_*, N_\&, N_\oplus)$ its AST, where the predicates C are defined analogous to the predicates in ASTs of global types.

We again identify V with the linear order of natural numbers produced by a DFS and denote with $\mathsf{action}(i)$ the action of the ith visited node. The auxiliary functions *spre* and *last* are also defined analogous and we refrain from denoting them with a different name for readability.

Definition 35 (Active Futures for Local Types)
The *act* function has the same intuition as in the global case, except not needing the object parameter and is defined as follows. The free variables in the side-conditions are implicitly quantified with an existential quantifier.

$$\begin{aligned}
\mathit{act} &: \mathbb{N} \rightharpoonup \mathsf{Fut}\\
\mathit{act}(0) &= \bot\\
\underset{i>0}{\mathit{act}(i)} &= \begin{cases} \bot & \text{if } \mathit{last}(\mathit{spre}(i)) = \mathsf{Await}(f, f')\\ \bot & \text{if } \mathit{last}(\mathit{spre}(i)) = \mathsf{Put}\ f\!:\!R\\ f & \text{if } \mathit{last}(\mathit{spre}(i)) = \mathbf{p}?_f m(R)\\ f & \text{if } \mathit{last}(\mathit{spre}(i)) = \mathsf{React}(f)\\ \mathit{act}(\mathit{last}(\mathit{spre}(i))) & \text{otherwise} \end{cases}
\end{aligned}$$

Again, $\mathit{last}(\mathit{spre}(i))$ is the id of the last action that must be executed before the action at i.

When projecting global to object-local types, not every constraint of the usage of futures is checked. To check these, especially branching and correct propagation, we define several auxiliary predicates.

Definition 36 (Knowing Futures)
The $\mathit{known} \subseteq \mathsf{Fut} \times \mathcal{P}(\mathsf{Fut}) \times \mathcal{P}(\mathbb{N})$ predicate models that a future f has access to every future in a set of futures S in a set of nodes I, i.e. f either received the future on its activating call or read it from a future with a Get.

$$\begin{aligned}
\mathit{known}(f, S, I) \iff \forall f' \in S.\ \exists R \in (\mathbb{N} \times \mathsf{Fut})^*.&\\
\Big(\big(\exists j \in I.\ \mathsf{action}(j) = \mathbf{q}?_f m(R) \wedge \exists k \in \mathbb{N}.\ (k, f') \in R\big)\vee&\\
\big(\exists j \in I.\ \exists f'' \in \mathsf{Fut}.\ \mathsf{action}(j) = \mathsf{Get}\ f''\!:\!R \wedge \exists k.\ (k, f') \in R\big)\vee&\\
\big(\exists j \in I.\ \mathsf{action}(j) = \mathbf{q}!_{f'} m(R)\big)\Big)&
\end{aligned}$$

Example 11
Consider the following global types

$$\mathbf{G}_1 = \mathbf{0} \xrightarrow{f_0} \mathbf{A}\!:\!m()\ .\ \mathbf{A} \xrightarrow{f} \mathbf{B}\!:\!m()\ .\ \mathbf{B} \xrightarrow{f'} \mathbf{C}\!:\!m()\ .\ \mathbf{C}{\downarrow}f'\ .\ \mathbf{B}{\downarrow}f\ .\ \mathbf{A}{\uparrow}f'\ .\ \mathbf{A}{\downarrow}f_0\ .\ \mathsf{end}$$

$$\mathbf{G}_2 = \mathbf{0} \xrightarrow{f_0} \mathbf{A}\!:\!m()\ .\ \mathbf{A} \xrightarrow{f} \mathbf{B}\!:\!m()\ .\ \mathbf{A} \xrightarrow{f'} \mathbf{C}\!:\!m()\ .\ \mathsf{Rel}(\mathbf{C}, f)\ .\ \mathbf{B}{\downarrow}f\ .\ \mathbf{C}{\downarrow}f'\ .\ \mathbf{A}{\uparrow}f'\ .\ \mathbf{A}{\downarrow}f_0\ .\ \mathsf{end}$$

Both are not propagating futures correctly.
In $\mathbf{G}_1$ the object $\mathbf{A}$ reads from f' but f' is only known to $\mathbf{B}$ and $\mathbf{B}$ does not communicate f' in its return value. In $\mathbf{G}_2$ the object $\mathbf{C}$ suspends its process until f is resolved, but f is only known to $\mathbf{A}$ which did not communicate it when calling $\mathbf{C}.m$. The following types are propagating futures correctly:

$$\mathbf{G}'_1 = \mathbf{0} \xrightarrow{f_0} \mathbf{A}\!:\!m()\ .\ \mathbf{A} \xrightarrow{f} \mathbf{B}\!:\!m()\ .\ \mathbf{B} \xrightarrow{f'} \mathbf{C}\!:\!m()\ .\ \mathbf{C}\!\downarrow\! f'\ .\ \underbrace{\mathbf{B}\!\downarrow\! f\!:\!(\bot, f')\ .\ \mathbf{A}\!\uparrow\! f\!:\!f'}_{\text{propagation}}\ .\ \mathbf{A}\!\uparrow\! f'\ .\ \mathbf{A}\!\downarrow\! f_0.\mathsf{end}$$

$$\mathbf{G}'_2 = \mathbf{0} \xrightarrow{f_0} \mathbf{A}\!:\!m()\ .\ \mathbf{A} \xrightarrow{f} \mathbf{B}\!:\!m()\ .\ \underbrace{\mathbf{A} \xrightarrow{f'} \mathbf{C}\!:\!m(\{0, f\})}_{\text{propagation}}\ .\ \mathsf{Rel}(\mathbf{C}, f)\ .\ \mathbf{B}\!\downarrow\! f\ .\ \mathbf{C}\!\downarrow\! f'\ .\ \mathbf{A}\!\uparrow\! f'\ .\ \mathbf{A}\!\downarrow\! f_0.\mathsf{end}$$

In $\mathbf{G}'_1$ the object $\mathbf{B}$ propagates f' back to $\mathbf{A}$ correctly. In $\mathbf{G}_2$ the object $\mathbf{A}$ propagates f correctly by passing it as a method parameter.

Definition 37 (Distinguishable Branches)
The *dist* $\subseteq \mathcal{P}(\mathsf{LT}) \times \mathbb{N}$ predicate models that every branch in a set $\{\mathbf{L}_j\}_{j\in J}$ is distinguishable from the others. Let i be the position of the branching in a fixed AST that has $\{\mathbf{L}_j\}_{j\in J}$ as the branches. For every future it either starts with a call or a resolving action, such that every call has another callee and every resolving action has another class label.

$$\begin{aligned}
&\mathit{dist}((\mathbf{L}_j)_{j\in J}) \iff \\
&\quad \exists J', J'' \subseteq J.\ J' \cap J'' = \emptyset \wedge J' \cup J'' = J \wedge \\
&\quad (\forall j \in J'.\ \exists \mathbf{L}'_j \in \mathsf{LT}.\ \mathbf{L}_j = \mathbf{q_j}!_{f_j} m_j(S_j)\ .\ \mathbf{L}'_j \wedge \bigwedge_{\substack{i,j\in J'\\ i\neq j}} (m_j \neq m_i \vee \mathbf{q}_j \neq \mathbf{q}_i) \wedge \\
&\quad (\forall j \in J''.\ \exists \mathbf{L}'_j \in \mathsf{LT}.\ \mathbf{L}_j = \mathsf{Put}\ f\!:\!(S_j, N_j)\ .\ \mathbf{L}'_j \wedge f = \mathit{act}(\mathit{child}(i,j))) \wedge \bigwedge_{\substack{i,j\in J''\\ i\neq j}} N_i \neq N_j)
\end{aligned}$$

A branching $\oplus\{\mathbf{L}_j\}_{j\in J}$ is distinguishable, if the set J can be partitioned into two parts J', J'', such that $\mathbf{L}_j, j \in J'$ are forward choices and $\mathbf{L}_j, j \in J''$ are backward choices. All branches in J' start with a call action and all branches in J'' start with a resolving action. The branches are distinguishable if every call at the start of a forward choice is to a different method or object and every call at the start of a backward choice uses a different constructor.

Note that after a forward choice a backward choice can be communicated. This is not expressed in the *dist* predicate but is captured in the check for well-formedness in Section 3.5.

Example 12
Consider the following object-local type

$$\mathbf{0}?_{f_0} \mathit{start}\ .\ \mathbf{B}!_f \mathit{reg}\ .\ \mathsf{Await}\ f_0, f\ .\ \&_f \left\{ \begin{array}{l} \mathbf{B}?_{f'} \mathit{more}\ .\ \mathsf{Put}\ f'\ .\ \mathsf{React}\ f_0\ .\ \mathsf{Get}\ f\!:\!\mathsf{ok}\ .\ \mathsf{Put}\ f_0.\mathsf{end} \\ \mathsf{React}\ f_0\ .\ \underbrace{\mathbf{C}!_{f''} \mathit{log}\ .\ \mathsf{Get}\ f''}_{\mathit{wrong}}\ .\ \mathsf{Get}\ f\!:\!\mathsf{deny}\ .\ \mathsf{Put}\ f_0.\mathsf{end} \end{array} \right\}$$

This is not correct because in the second branch the process computing f_0 calls $\mathbf{C}.\mathit{log}$ before being notified about the branch choice by reading from f''. It is however correct that in one branch the method *more* is called, because this is outside of the process reading from f. Consider the local type

$$\mathbf{0}?_{f_0} \mathit{start}\ .\ \mathbf{B}!_f \mathit{reg}\ .\ \mathsf{Await}\ f_0, f\ .\ \&_f \left\{ \begin{array}{l} \mathbf{B}?_{f'} \mathit{more}\ .\ \mathsf{Put}\ f'\ .\ \mathsf{React}\ f_0\ .\ \mathsf{Get}\ f\!:\!\mathsf{ok}\ .\ \mathsf{Put}\ f_0.\mathsf{end} \\ \mathsf{React}\ f_0\ .\ \mathsf{Get}\ f\!:\!\mathsf{ok}\ .\ \mathbf{C}!_{f''} \mathit{log}\ .\ \mathsf{Get}\ f''\ .\ \mathsf{Put}\ f_0.\mathsf{end} \\ \mathsf{React}\ f_0\ .\ \mathsf{Get}\ f\!:\!\mathsf{deny}\ .\ \mathbf{C}!_{f'''} \mathit{log}\ .\ \mathsf{Get}\ f'''\ .\ \mathsf{Put}\ f_0.\mathsf{end} \end{array} \right\}$$

This is not correct because there are two branches where $\mathbf{B}$ communicates ok: the process computing f_0 behaves differently in these branches despite not knowing which of them was chosen as the information is only communicated via f and $\mathbf{B}$ does not access it.. Note that object $\mathbf{A}$ can distinguish between them, if the process computing f' would store the information that it was called on *more*. The following type is propagating futures correctly:

$$\mathbf{0}?_{f_0}start\ .\ \mathbf{B}!_f reg\ .\ \mathsf{Await}\ f_0, f\ .\ \&_f \left\{ \begin{array}{l} \mathbf{B}?_{f'}more\ .\ \mathsf{Put}\ f'\ .\ \mathsf{React}\ f_0\ .\ \mathsf{Get}\ f\!:\!\mathsf{ok}\ .\ \mathsf{Put}\ f_0.\mathsf{end} \\ \mathsf{React}\ f_0\ .\ \mathsf{Get}\ f\!:\!\mathsf{ok}\ .\ \mathsf{Put}\ f_0.\mathsf{end} \\ \mathsf{React}\ f_0\ .\ \mathsf{Get}\ f\!:\!\mathsf{deny}\ .\ \mathbf{C}!_{f''}log\ .\ \mathsf{Get}\ f''\ .\ \mathsf{Put}\ f_0.\mathsf{end} \end{array} \right\}$$

Also the set of constructors needs not be exhaustive, i.e. there may be constructor names which are not used to communicate choice. We ignore these, as their use is not described in the protocol. We verify however that they are not used by the method during verification of the method code.

Additionally, the $fresh \subseteq \mathsf{Fut} \times \mathcal{P}(\mathbb{N})$ predicate models that a future f was not active in the nodes I. If a future was not active for $spre(i)$, then it is fresh.

$$fresh(f, I) \iff \forall i \in I.\ act(i) \neq f$$

3.3. Projection

Projection is the procedure to derive the *local* type of an endpoint on a session type. In ABS, the notion of an endpoint is twofold: data is sent between processes of different objects. We use a two-fold projection: First the global type is projected onto an object and the resulting object-local type describes the behavior of this object within the system. Secondly the object-local type is projected onto a future and the resulting method-local type describes the behavior of the process computing this future.

So far, we gave no notion of *Well-Formedness* for session types. However not every syntactically correct type is a valid description of an ABS system. We check this during the projection; a global type is well-formed if every projection is defined.

Additional concepts are needed to define well-formed types and formalize the descriptions we presented in the last sections.

Actions on futures can not be repeated arbitrarily. For each invocation with a future there is exactly one invocation reaction and at most one resolving event. We also demand that there is exactly one resolving event, i.e. the type describes the whole communication and at the end of the session all processes participating in it have terminated.

A type is self-contained, if it resolves every future which is started in it. Only self-contained types can be repeated.

Definition 38 (Self-Contained Types)
A global type $\mathbf{G}$ is *self-contained* if

- for every call action within $\mathbf{G}$, there is a corresponding resolving action within $\mathbf{G}$,
- for every resolving action within $\mathbf{G}$, there is a corresponding call action within $\mathbf{G}$,
- it contains no end,
- and each repetition within $\mathbf{G}$ is self-contained

A local type $\mathbf{L}$ is *self-contained* if

- for every reactivation within $\mathbf{L}$, there is a corresponding suspending action within $\mathbf{L}$
- for every suspending action within $\mathbf{L}$, there is a corresponding reactivation within $\mathbf{L}$
- for every receiving action within $\mathbf{L}$, there is a corresponding resolving action within $\mathbf{L}$
- for every resolving action within $\mathbf{L}$, there is a corresponding receiving action within $\mathbf{L}$
- it contains no end,
- and each repetition within $\mathbf{L}$ is self-contained

Example 13
The following type is not self-contained, because inside the repetition there is no corresponding call for the resolving action

$$\mathbf{A} \xrightarrow{f} \mathbf{B} \ . \ (\mathbf{B}\downarrow f)^*$$

The following type is self-contained

$$(\mathbf{A} \xrightarrow{f} \mathbf{B} \ . \ \mathbf{B}\downarrow f)^*$$

A self-contained type resolves exactly his own futures. We only allow self-contained types to be repeated. This way it is ensured that every constraint on futures that demands that something happens exactly once (one invocation, once invocation reaction, etc.) is adhered to.

3.3.1. Projecting Global to Object-Local Types

We define the projection of a global type $\mathbf{G}$ to the local type of an object $\mathbf{p}$ as a *partial* function $\mathbf{G}\restriction_{\mathbf{p},i}$ where i denotes that $\mathbf{G}$ is the ith node in a fixed, given AST and allows to use information from the *act* and *wait* functions of the prefix of a type.

The projection is defined with a case distinction on the form of $\mathbf{G}$. If there is no case whose side-condition is fulfilled, the projection is undefined. Some of the side-conditions are needed to model that futures are used correctly, but some violations are covered in the projection to method-local types. I.e. if the global type does not use futures correctly, $\mathbf{G}\restriction_{\mathbf{p},0}$ may be defined despite not using futures correctly. This is not problematic as only the method-local types are checked against ABS code. The checks on this level ensure correct usage of futures across multiple objects, i.e. that communication between objects does not lead to ill-formed communication histories. The side-conditions while projecting the object-local type to method-local types ensure the correct usage of future within a single object.

The projection $\mathbf{G}\restriction_{\mathbf{p},i}$ is defined in Figure 3.2, using the $wait(i, f)$ function to model which futures are waiting for f, the $fresh(f, S)$ predicate to model that a future f is fresh and the $reads(\mathbf{G}, f)$ function to model that f is read in $\mathbf{G}$. The function are defined in Section 3.1.

- The call action is mapped to a receive action on the callee side and a sending action on the caller side.
- A resolving action is mapped to a putting action of the corresponding object, to a reactivation for every waiting object and skip for any other object.

- A read action is mapped to a getting action of the corresponding object and skip for any other object. The side-condition ensures that a future is resolved before the future is read and only futures passed during resolving is read. This must be checked at a global level, because resolving and reading takes place in different objects.
- A release action is mapped to a suspending action of the corresponding object and skip for any other object. The side-condition ensures that the releasing object does not have any other future waiting for the same resolving. This is needed because we can not reason about the scheduling of ABS and must be sure that the correct process is reactivated.
- Branching is mapped to choice for the active side and offer for every side that either receives one of the calls publishing the choice or reads from the active future. For any other object there must be a unique local type that is equivalent in every branch, i.e. an object that does not receive the choice behaves the same. If objects were allowed to behave differently, it must be verified that they behave correctly in a given situation. We can identify the situation and the knowledge of a process only by its future accesses. If a process has the same information in two position in a session types and behaves differently, we can not ensure that the correct choice *how* to behave is made.
- Termination is mapped to termination. The condition that every future has been resolved, i.e. all objects are inactive and no future is still suspended, is not checked during projection but as an extra condition for well-formedness.
- Repetition and concatenation are propagated down.

$$
\mathbf{r} \xrightarrow{f} \mathbf{q}{:}m(R) \upharpoonright_{\mathbf{p},i} = \begin{cases} \mathbf{r}!_f m(R) & \text{if } \mathbf{p} = \mathbf{r} \wedge \mathit{fresh}(f,i) \\ \mathbf{q}?_f m(R) & \text{if } \mathbf{p} = \mathbf{q} \wedge \mathit{fresh}(f,i) \\ \mathsf{skip} & \text{if } \mathbf{p} \neq \mathbf{q} \wedge \mathbf{p} \neq \mathbf{r} \wedge \mathit{fresh}(f,i) \end{cases}
$$

$$
\mathbf{q}{\downarrow} f : (C,R) \upharpoonright_{\mathbf{p},i} = \begin{cases} \mathsf{Put}\ f : (C,R) & \text{if } \mathbf{q} = \mathbf{p} \\ \mathsf{React}(f') & \text{if } \mathbf{q} \neq \mathbf{p} \wedge (\mathbf{p}, f') \in \mathit{wait}(i,f) \\ \mathsf{skip} & \text{otherwise} \end{cases}
$$

$$
\mathbf{q}{\uparrow} f : (C,R) \upharpoonright_{\mathbf{p},i} = \begin{cases} \mathsf{Get}\ f : (C,R) & \text{if } \mathbf{q} = \mathbf{p} \wedge \exists j \in \mathit{spre}(i).\ \mathsf{action}(j) = \mathbf{r}{\downarrow} f : (C,R') \wedge R \subseteq R' \\ \mathsf{skip} & \text{if } \exists j \in \mathit{spre}(i).\ \mathsf{action}(j) = \mathbf{r}{\downarrow} f : (C',R') \wedge C' \preceq C \end{cases}
$$

$$
\mathsf{Rel}(\mathbf{q}, f) \upharpoonright_{\mathbf{p},i} = \begin{cases} \mathsf{Await}(\mathit{act}(i,\mathbf{q}), f) & \text{if } \mathbf{q} = \mathbf{p} \wedge\ \nexists f' \in \mathsf{Fut}.(\mathbf{q}, f') \in \mathit{wait}(i,f) \\ \mathsf{skip} & \text{if } \ \nexists f' \in \mathsf{Fut}.(\mathbf{q}, f') \in \mathit{wait}(i,f) \end{cases}
$$

$$
\mathbf{q}\{\mathbf{G}_j\}_{j \in J} \upharpoonright_{\mathbf{p},i} = \begin{cases} \oplus\{\mathbf{G}_j \upharpoonright_{O,\mathit{child}(i,j)}\}_{j \in J} & \text{if } \mathbf{p} = \mathbf{q} \\ \&_f\{\mathbf{G}_j \upharpoonright_{O,\mathit{child}(i,j)}\}_{j \in J} & \text{if } \mathbf{p} \neq \mathbf{q} \wedge \exists j \in J.\ \mathbf{p} \in \mathit{reads}(\mathbf{G}_j, f) \wedge f = \mathit{act}(i,\mathbf{q}) \\ \mathbf{G} & \text{if } \forall j \in J.\ \mathbf{G}_j \upharpoonright_{\mathbf{p},\mathit{child}(i,j)} = \mathbf{G} \end{cases}
$$

$$
\begin{aligned}
\mathsf{end} \upharpoonright_{\mathbf{p},i} &= \mathsf{end} \\
(\mathbf{G}_1\ .\ \mathbf{G}_2) \upharpoonright_{\mathbf{p},i} &= \mathbf{G}_1 \upharpoonright_{\mathbf{p},\mathit{child}(i,1)}\ .\ \mathbf{G}_2 \upharpoonright_{\mathbf{p},\mathit{child}(i,2)} \\
\mathbf{G}^* \upharpoonright_{\mathbf{p},i} &= (\mathbf{G} \upharpoonright_{\mathbf{p},\mathit{child}(i,1)})^*
\end{aligned}
$$

Figure 3.2.: Projection of Global to Object-Local Types

We write $\mathbf{G}\restriction_{\mathbf{p}}$ for $\mathbf{G}\restriction_{\mathbf{p},0}$.

Example 14
Consider the global type from Example 16:

$$\mathsf{Q} = \mathbf{0} \xrightarrow{f_0} \mathbf{A}\!:\!start\ .\ \left(\mathbf{A}\left\{\begin{array}{l}\mathbf{A} \xrightarrow{f'} \mathbf{B}\!:\!m\ .\ \mathbf{B}{\downarrow}f'\ .\ \mathbf{A}{\uparrow}f' \\ \mathbf{A} \xrightarrow{f''} \mathbf{C}\!:\!m\ .\ \mathbf{C}{\downarrow}f''\ .\ \mathbf{A}{\uparrow}f''\end{array}\right\}\right)^{*}\ .\ \mathbf{A}{\downarrow}f_0\ .\ \mathsf{end}$$

Its projection on $\mathbf{A}$ is

$$\mathsf{Q}\restriction_{\mathbf{A}} = \mathbf{0}?_{f_0}start\ .\ \left(\oplus\left\{\begin{array}{l}\mathbf{B}!_{f'}m\ .\ \mathsf{Get}\ f' \\ \mathbf{C}!_{f''}m\ .\ \mathsf{Get}\ f''\end{array}\right\}\right)^{*}\ .\ \mathsf{Put}\ f_0\ .\ \mathsf{end}$$

The projection on $\mathbf{B}$ is

$$\mathsf{Q}\restriction_{\mathbf{B}} = \left(\mathbf{A}?_{f'}m\ .\ \mathsf{Put}\ f'\right)^{*}\ .\ \mathsf{end}$$

The projection on $\mathbf{C}$ is

$$\mathsf{Q}\restriction_{\mathbf{C}} = \left(\mathbf{A}?_{f''}m\ .\ \mathsf{Put}\ f''\right)^{*}\ .\ \mathsf{end}$$

Note the repetition in the local types of **B** and **C**. This repetition is outside of any process and not visible from the view of a single process. It is however visible from the view of the whole object as *being repeatedly called*. The termination is also outside of each process.

3.3.2. Projecting Object-Local to Method-Local Types

We define the projection of a local type $\mathbf{L}$ to the local type of a future f as a *partial* function $\mathbf{L}\restriction_{f,i}$ where i denotes that $\mathbf{L}$ is the ith node and allows to use information from the *act* function of the prefix of a type.

The projection is defined with a case distinction on the form of $\mathbf{L}$. If there is no case whose side-condition is fulfilled, the projection is undefined. Let p be a fixed object. The definition of $\mathbf{L}\restriction_{f,i}$ is given in Figure 3.3

- A sending action is mapped to itself for the active future and to skip for any other. The side-conditions ensure that only futures which are known to the active future are sent, the new future is fresh and there is an active future.

- A receiving action is mapped to itself for the received future and to skip for any other. The side-conditions ensure that there is no active future.

- A resolving action is mapped to itself for the active future and to skip for any other. The side-conditions ensure that only futures which are known to the active future are sent and there is an active future.

- A fetching action is mapped to itself for the active future and to skip for any other. The side-conditions ensure that the active future knows the future it reads from and there is an active future.

- A suspending action is mapped to itself for the active future and to skip for any other. The side-conditions ensure that the active future knows the future it waits for and there is an active future.

$$\mathbf{q}!_f m(R) \restriction_{f',i} = \begin{cases} \mathbf{q}!_f m(R) & \text{if } act(i) = f = f' \wedge fresh(f, spre(i)) \wedge known\big(act(i), futs(R), spre(i)\big) \\ \mathsf{skip} & \text{if } act(i) \neq f \wedge fresh(f, spre(i)) \wedge act(i) \neq \bot \end{cases}$$

$$\mathbf{q}?_{f'} m(R) \restriction_{f,i} = \begin{cases} \mathbf{q}?_f m(R) & \text{if } f' = f \wedge \neg fresh(f, spre(i)) \wedge act(i) = \bot \\ \mathsf{skip} & \text{if } f' \neq f \wedge act(i) = \bot \end{cases}$$

$$\mathsf{Put}\ f' {:} (C, R) \restriction_{f,i} = \begin{cases} \mathsf{Put}\ f' {:} (C, R) & \text{if } act(i) = f = f' \wedge known\big(act(i), futs(R), spre(i)\big) \\ \mathsf{skip} & \text{if } act(i) \neq f \wedge act(i) \neq \bot \end{cases}$$

$$\mathsf{Get}\ f' {:} (C, R) \restriction_{f,i} = \begin{cases} \mathsf{Get}\ f' {:} (C, R) & \text{if } act(i) = f \wedge known\big(act(i), \{f\}, spre(i)\big) \\ \mathsf{skip} & \text{if } act(i) \neq f \wedge act(i) \neq \bot \end{cases}$$

$$\mathsf{Await}(f', f'') \restriction_{f,i} = \begin{cases} \mathsf{Await}(f', f'') & \text{if } act(i) = f = f' \wedge known\big(act(i), \{f''\}, spre(i)\big) \\ \mathsf{skip} & \text{if } act(i) \neq f' \wedge act(i) \neq \bot \end{cases}$$

$$\oplus\{\mathbf{L}_j\}_{j \in J} \restriction_{f,i} = \begin{cases} \oplus\{\mathbf{L}_j \restriction_{f,child(i,j)}\}_{j \in J} & \text{if } act(i) = f \wedge dist((\mathbf{L}_j)_{j \in J}) \\ \mathbf{L} & \text{if } act(i) \neq f \wedge \neg fresh(f, subs(i+j)) \wedge dist((\mathbf{L}_j)_{j \in J}) \wedge \\ & \quad \big((\forall j \in J.\ \mathbf{L}_j \restriction_{f,child(i,j)} = \mathbf{L}) \vee \\ & \quad (\exists k \in J\ \mathbf{L}_k \restriction_{f,child(i,k)} = \mathbf{L} \\ & \quad \wedge \forall j \in J.\ j \neq k \rightarrow \mathbf{L}_j \restriction_{f,child(i,j)} = \mathsf{skip})\big) \\ \mathsf{skip} & \text{if } act(i) \neq f \wedge fresh(f, subs(i+j)) \wedge dist((\mathbf{L}_j)_{j \in J}) \end{cases}$$

$$\&_{f'}\{\mathbf{L}_j\}_{j \in J} \restriction_{f,i} = \begin{cases} \mathbf{L} & \text{if } (\forall j \in J.\ \mathbf{L}_j \restriction_{f,child(i,j)} = \mathbf{L}) \vee \\ & \quad (\exists k \in J\ \mathbf{L}_k \restriction_{f,child(i,k)} = \mathbf{L} \\ & \quad \wedge \forall j \in J.\ j \neq k \rightarrow \mathbf{L}_j \restriction_{f,child(i,j)} = \mathsf{skip}) \\ \&_{f'}\{\mathbf{L}_j \restriction_{f,child(i,j)}\}_{j \in J} & \text{if } \Big(\exists C_1, \ldots, C_{|J|} \in \mathsf{Con}. \\ & \quad \big((\forall j \in J.\ \mathbf{L}_j \restriction_{f,child(i,j)} = \mathsf{Get}\ f' {:} C_j\ .\ \mathbf{L}'_j) \vee \\ & \quad (\forall j \in J.\ \mathbf{L}_j \restriction_{f,child(i,j)} = \mathsf{React}(f)\ .\ \mathsf{Get}\ f' {:} C_j\ .\ \mathbf{L}'_j)\big) \wedge \\ & \quad \forall j, k \in J.\ C_j = C_k \rightarrow \mathbf{L}_j \restriction_{f,child(i,j)} \equiv_{\mathsf{Fut}} \mathbf{L}_k \restriction_{f,child(i,k)} \Big) \end{cases}$$

$$\mathbf{L}^* \restriction_{f,i} = \begin{cases} (\mathbf{L} \restriction_{f,i+1})^* & \text{if } \neg fresh(f, spre(i)) \wedge \mathbf{L} \text{ is closed} \\ \mathbf{L} \restriction_{f,i+1} & \text{if } fresh(f, spre(i)) \wedge \mathbf{L} \text{ is closed} \end{cases}$$

$$\begin{aligned} \mathsf{React}(f) \restriction_{f',i} &= \mathsf{skip} \text{ if } act(i) = \bot \wedge \mathsf{action}\big(\max\{j \mid j \in spre(i) \wedge act(j,p) = f\}\big) = \mathsf{Await}(f, f'') \\ \mathsf{end} \restriction_{f,i} &= \mathsf{skip} \\ (\mathbf{L}_1\ .\ \mathbf{L}_2) \restriction_{f,i} &= \mathbf{L}_1 \restriction_{f,child(i,2)}\ .\ \mathbf{L}_2 \restriction_{f,child(i,2)} \\ \mathsf{skip} \restriction_{f,i} &= \mathsf{skip} \end{aligned}$$

Figure 3.3.: Projection of Object-Local to Method-Local Types

- A reactivation action is mapped to skip for any future, because in the setting of partial correctness the next action after a suspending action will always be a reactivation inside a method-local type. The side-condition ensures that there is no active future and the last action of the reactivated future was a suspending action.
- A choice action is mapped
  - to a choice for the active future,
  - to the unique $\mathbf{L}$ that it is mapped to in every branch for suspended futures,
  - to the unique $\mathbf{L}$ that it is mapped to for futures which are fresh but used in exactly one of the branches and
  - to skip for any other future.

  The side-condition ensures that the branches are distinguishable.
- A offer action is mapped
  - to the unique $\mathbf{L}$ that it is mapped to in every branch for any future which does not read from the future carrying the branch choice
  - to an offer action for any future that does

  The side-condition ensures that the branches are distinguishable and each future is mapped to the same type up to the point where it reads the choice.
- Repetition is mapped to a repetition for futures which are active or suspended, and to the simple projection for any other futures, because the repetition is being repeated called from their view.
- Termination is always mapped to skip because it is not visible
- Concatenation and the empty type are handled analogous to their global counterpart.

Example 15
Consider the following type:

$$\mathbf{0} \xrightarrow{f_0} \mathbf{A}\!:\!\mathit{start}.\left(\mathbf{A} \xrightarrow{f} \mathbf{B}\!:\!\mathit{reg}.\mathsf{Rel}(\mathbf{A}, f_0).\mathbf{B}\left\{\begin{array}{l} \mathbf{B} \xrightarrow{f'} \mathbf{A}\!:\!\mathit{more}\ .\ \mathbf{A}{\downarrow}f'\ .\ \mathbf{B}{\uparrow}f'\ .\ \mathbf{B}{\downarrow}f\!:\!\mathsf{ok}\ .\ \mathbf{A}{\uparrow}f \\ \mathbf{B}{\downarrow}f\!:\!\mathsf{deny}\ .\ \mathbf{A}{\uparrow}f\ .\ \mathbf{A} \xrightarrow{f''} \mathbf{C}\!:\!\mathit{log}\ .\ \mathbf{C}{\downarrow}f''\ .\ \mathbf{A}{\uparrow}f'' \end{array}\right\}\right)^{*}.\mathbf{A}{\downarrow}f_0.\mathsf{end}$$

The object-local type of $\mathbf{A}$ is:

$$\mathbf{0}?_{f_0}\mathit{start}.\left(\mathbf{B}!_f\mathit{reg}\ .\ \mathsf{Await}\ (f_0, f)\ .\ \&_f\left\{\begin{array}{l} \mathbf{B}?_{f'}\mathit{more}\ .\ \mathsf{Put}\ f'\ .\ \mathsf{React}(f_0)\ .\ \mathsf{Get}\ f\!:\!\mathsf{ok} \\ \mathsf{React}(f_0)\ .\ \mathsf{Get}\ f\!:\!\mathsf{deny}\ .\ \mathbf{C}!_{f''}\mathit{log}\ .\ \mathsf{Get}\ f'' \end{array}\right\}\right)^{*}.\ \mathsf{Put}\ f_0\ .\ \mathsf{end}$$

The method-local type of *start*:

$$\mathbf{0}?_{f_0}\mathit{start}\ .\ \left(\mathbf{B}!_f\mathit{reg}\ .\ \mathsf{Await}\ (f_0, f)\ .\ \&_f\left\{\begin{array}{l} \mathsf{React}(f_0)\ .\ \mathsf{Get}\ f\!:\!\mathsf{ok} \\ \mathsf{React}(f_0)\ .\ \mathsf{Get}\ f\!:\!\mathsf{deny}\ .\ \mathbf{C}!_{f''}\mathit{log}\ .\ \mathsf{Get}\ f'' \end{array}\right\}\right)^{*}.\ \mathsf{Put}\ f_0$$

The object-local type of $\mathbf{C}$:

$$(\mathbf{A}?_{f''}\mathit{log}\ .\ \mathsf{Put}\ f'')^{*}\ .\ \mathsf{end}$$

The method-local type of *log*

$$\mathbf{A}?_{f''}\mathit{log}\ .\ \mathsf{Put}\ f''$$

Note that on the object-level, the repeated call of *log* is visible to $\mathbf{C}$. It is not visible to a single process computing *log*. Also the object-local type of $\mathbf{C}$ does not contain a branching, because during projection, all but one branch are equal to skip.

3.4. Translation of Types to Regular Expressions

As we regard session types as propositions over sequences, we translate types into *regular expressions*. The language of such an expression is the set of histories realized by a system under simplistic assumptions like having no delay.

The transition to the real concurrency model of ABS is made by allowing *permutations*: it is not checked whether the history a system realized is described by the regular expression of a type; instead it is checked whether it is some permutation of such a history. Additionally the local history of all objects must be the same in the realized and the described history - from a local point of view, the simple and the real system are not distinguishable.

We are also able to formalize what it means that a type only describes possible executions of systems: all histories described by its translation into a regular expression are well-formed.

3.4.1. Translation of Global Types to Regular Expressions

Definition 39 (Translation of Global Types to Regular Expressions)
The translation $t(C, R)$ translates a constructor C and a set of pairs of futures and positions R into the corresponding expression. If $C = \bot$ then $t(\bot, R) = \textbf{null}$, otherwise for $C \neq \bot$:

$$t(C, R) = C(j_1, \ldots, j_{ar(C)}) \text{ with } j_n = \begin{cases} f & \text{if } (n, f) \in R \\ \textbf{null} & \text{otherwise} \end{cases}$$

The translation $t(m, R)$ translates a set of pairs of futures and position into the corresponding list of parameters for a method m.

$$t(m, R) = [j_1, \ldots, j_{ar(m)}] \text{ with } j_n = \begin{cases} f & \text{if } (n, f) \in R \\ \textbf{null} & \text{otherwise} \end{cases}$$

The global type translator $\tau : \mathsf{GT} \to \mathcal{P}(\mathsf{Ev}^*)$ maps a global type to a regular expression. Let $S_{\mathbf{G}} : \mathsf{Fut} \to \mathsf{Met}$ be the mapping from futures to the corresponding method in $\mathbf{G}$, i.e. $S_{\mathbf{G}}(f) = m$ if $\mathbf{p} \xrightarrow{f} \mathbf{q}{:}m$ occurs somewhere in $\mathbf{G}$. We assume a fixed $S_{\mathbf{G}}$ function.

Then τ is defined by:

$$\tau(\mathbf{p} \xrightarrow{f} \mathbf{q}{:}m(R)) = \begin{cases} [\mathsf{invEv}(\mathbf{p}, \mathbf{q}, f, m, t(m, R)), \mathsf{invREv}(\mathbf{p}, \mathbf{q}, f, m, t(m, R))] & \text{if } \mathbf{p} \neq \mathbf{0} \\ [\mathsf{invREv}(\mathbf{p}, \mathbf{q}, f, m, t(m, R))] & \text{if } \mathbf{p} = \mathbf{0} \end{cases}$$

$$\tau(\mathbf{p}{\downarrow} f : (C, R)) = \begin{cases} [\mathsf{futEv}(\mathbf{p}, f, S_{\mathbf{G}}(f), t(C, R))] & \text{if } C \text{ is not an Exception} \\ [\mathsf{throwEv}(\mathbf{p}, f, S_{\mathbf{G}}(f), t(C, R))] & \text{if } C \text{ is an Exception} \end{cases}$$

$$\tau(\mathbf{q}{\uparrow} f : (C, R)) = \begin{cases} [\mathsf{futREv}(\mathbf{q}, f, t(C, R))] & \text{if } C \text{ is not an Exception} \\ [\mathsf{throwREv}(\mathbf{q}, f, t(C, R))] & \text{if } C \text{ is an Exception} \end{cases}$$

$$\begin{aligned} \tau(\mathsf{Rel}(\mathbf{p}, f)) &= [\mathsf{awaitEv}(\mathbf{p}, act(i, \mathbf{p}), f)] \text{ where } i \text{ is the position of this action in } \mathfrak{T}(\mathbf{G}) \\ \tau(\mathbf{G}_1 \,.\, \mathbf{G}_2) &= \tau(\mathbf{G}_1) \circ \tau(\mathbf{G}_2) \\ \tau(\mathbf{p}\{\mathbf{G}_j\}_{j \in J}) &= \tau(\mathbf{G}_i) + \tau(\mathbf{p}\{\mathbf{G}_j\}_{j \in J \setminus \{i\}}) \text{ with } i \in J \\ \tau(\mathbf{G}^*) &= \tau(\mathbf{G})^* \\ \tau(\mathsf{end}) &= \epsilon \end{aligned}$$

Example 16
Consider the following global type:

$$\mathsf{Q} = \mathbf{0} \xrightarrow{f_0} \mathbf{A}{:}\,start\ .\ \left(\mathbf{A}\left\{\begin{array}{l}\mathbf{A} \xrightarrow{f'} \mathbf{B}{:}\,m\ .\ \mathbf{B}{\downarrow}f'\ .\ \mathbf{A}{\uparrow}f' \\ \mathbf{A} \xrightarrow{f''} \mathbf{C}{:}\,m\ .\ \mathbf{C}{\downarrow}f''\ .\ \mathbf{A}{\uparrow}f''\end{array}\right\}\right)^{*}\ .\ \mathbf{A}{\downarrow}f_0\ .\ \mathsf{end}$$

Its translation is

$$\begin{aligned}\tau(\mathsf{Q}) = &\big[\mathsf{invREv}(0, \mathbf{A}, f_0, start, \epsilon)\big]\circ \\ &\Big(\big[\mathsf{invEv}(\mathbf{A}, \mathbf{B}, f', m, \epsilon), \mathsf{invREv}(\mathbf{A}, \mathbf{B}, f', m, \epsilon)\big]\circ \\ &\ \big[\mathsf{futEv}(\mathbf{B}, f', m, \mathbf{null}), \mathsf{futREv}(\mathbf{A}, f', \mathbf{null})\big] \\ &+ \\ &\big[\mathsf{invEv}(\mathbf{A}, \mathbf{C}, f'', m, \epsilon), \mathsf{invREv}(\mathbf{A}, \mathbf{C}, f'', m, \epsilon)\big]\circ \\ &\big[\mathsf{futEv}(\mathbf{C}, f'', m, \mathbf{null}), \mathsf{futREv}(\mathbf{A}, f'', \mathbf{null})\big]\Big)^{*} \\ &\circ \big[\mathsf{futEv}(\mathbf{A}, f_0, start, \mathbf{null})\big]\end{aligned}$$

Histories capture more information about method parameters and return values than session types can express. To compare histories generated by a system and histories described by a session type, we erase the additional parameters.

Definition 40 (Expression-Cleaned Histories)
Each event that carries data, has an *expr* parameter for the carried data. Let h be a history. The expression-cleaned history $\tilde{h}$ results from replacing every *expr* parameter with the encoding of future propagation. If *expr* has an algebraic data type and its outermost constructor is C, it is replaced by a value whose outermost constructor is C and every non-future argument is replaced by null. If *expr* is a list of method parameters this is done for each element. If no element contains a future the list is replaced by ϵ, if a single element does not contain any future, then this parameter is replaced by **null**. Otherwise *expr* is replaced by **null**.

To incorporate concurrency, a global history is captured by a type if it is the permutation of a history described by the translation.

Definition 41 (Capturing Histories)
A well-formed global history h is captured by global type $\mathbf{G}$, if $\tilde{h}$ is future-equivalent to a permutation of a history described by $\tau(G)$:

$$h{:}\mathbf{G} \iff \exists h', h'' \in \mathsf{Ev}^*.\ \tilde{h} \equiv_{\mathsf{Fut}} h' \wedge \pi(h', h'') \wedge h'' \in \mathcal{L}(\tau(\mathbf{G})) \wedge \forall \mathbf{p} \in \mathsf{Ob}.\ h\restriction_{\mathbf{p}} = h''\restriction_{\mathbf{p}}$$

3.4.2. Translation of Local Types into Regular Expressions

Analogous to global types we define a translation to regular expressions and semantics by permutations.

Definition 42 (Translation of Local Types into Regular Expressions)
The translation $\tau_{\mathbf{p}} : \mathsf{LT} \times (\mathsf{Fut} \to \mathsf{Met}) \to \mathcal{P}(\mathsf{Ev}^*)$ maps a local type for $\mathbf{p}$ to a regular expression. Let $S_{\mathbf{L}} : \mathsf{Fut} \to \mathsf{Met}$ be the mapping from futures to the corresponding method in

$\mathbf{L}$, i.e. $S_{\mathbf{L}}(f) = m$ if $\mathbf{p}?_f m(R)$ occurs somewhere in $\mathbf{L}$. Then $\tau_{\mathbf{p}}$ is defined by:

$$\begin{aligned}
\tau_{\mathbf{p}}(\mathbf{q}?_f m(R)) &= \big[\mathsf{invREv}(\mathbf{q}, \mathbf{p}, f, m, t(m, R))\big] \\
\tau_{\mathbf{p}}(\mathbf{q}!_f m(R)) &= \mathsf{invEv}(\mathbf{p}, \mathbf{q}, f, m, t(m, R)) \\
\tau_{\mathbf{p}}(\mathsf{Put}\ f\colon(C, R)) &= \begin{cases} \big[\mathsf{futEv}(\mathbf{p}, f, S_{\mathbf{L}}(f), t(C, R))\big] & \text{if } C \text{ is not an Exception} \\ \big[\mathsf{throwEv}(\mathbf{p}, f, S_{\mathbf{L}}(f), t(C, R))\big] & \text{if } C \text{ is an Exception} \end{cases} \\
\tau_{\mathbf{p}}(\mathsf{Get}\ f\colon(C, R)) &= \begin{cases} \big[\mathsf{futREv}(\mathbf{p}, f, t(C, R))\big] & \text{if } C \text{ is not an Exception} \\ \big[\mathsf{throwREv}(\mathbf{p}, f, t(C, R))\big] & \text{if } C \text{ is an Exception} \end{cases} \\
\tau_{\mathbf{p}}(\mathsf{Await}(f, f')) &= [\mathsf{awaitEv}(\mathbf{p}, f, f')] \\
\tau_{\mathbf{p}}(\mathbf{L}_1\ .\ \mathbf{L}_2) &= \tau_{\mathbf{p}}(\mathbf{L}_1) \circ \tau_{\mathbf{p}}(\mathbf{L}_2) \\
\tau_{\mathbf{p}}(\oplus\{\mathbf{L}_j\}_{j\in J}) &= \begin{cases} \tau_{\mathbf{p}}(\mathbf{L}_i) + \tau_{\mathbf{p}}(\oplus\{\mathbf{L}_j\}_{j\in J\setminus\{i\}}) \text{ with } i \in J & \text{if } |J| > 1 \\ \tau_{\mathbf{p}}(\mathbf{L}_i) & \text{if } J = \{i\} \end{cases} \\
\tau_{\mathbf{p}}(\&_f\{\mathbf{L}_j\}_{j\in J}) &= \begin{cases} \tau_{\mathbf{p}}(\mathbf{L}_i) + \tau_{\mathbf{p}}(\&_f\{\mathbf{L}_j\}_{j\in J\setminus\{i\}}) \text{ with } i \in J & \text{if } |J| > 1 \\ \tau_{\mathbf{p}}(\mathbf{L}_i) & \text{if } J = \{i\} \end{cases} \\
\tau_{\mathbf{p}}(\mathbf{L}^*) &= \tau_{\mathbf{p}}(\mathbf{L})^* \\
\tau_{\mathbf{p}}(\mathsf{skip}) &= \epsilon \\
\tau_{\mathbf{p}}(\mathsf{React}(f)) &= \epsilon \\
\tau_{\mathbf{p}}(\mathsf{end}) &= \epsilon
\end{aligned}$$

The translation of a type $\mathbf{L}$ is $\tau_{\mathbf{p}}(\mathbf{L})$.

Definition 43 (Capturing Local Histories)
A well-formed local history h for an object $\mathbf{p}$ is captured with a local type $\mathbf{L}$, if $\tilde{h}$ is future-equivalent to a history described by $\tau_{\mathbf{p}}(\mathbf{L})$:

$$h\colon\mathbf{L} \iff \exists h' \in \mathsf{Ev}^*.\ \tilde{h} \equiv_{\mathsf{Fut}} h' \wedge h' \in \mathcal{L}(\tau_{\mathbf{p}}(\mathbf{L}))$$

Permutations are *not* needed here, because on the object and process level we model the communication without delays and asynchronicity.

Definition 44 (Capturing Method-Local Types)
Let $\mathbf{L}$ be a mthod-local type and m a method. The method m ist caputered by $\mathbf{L}$ if every history that m can realize is captured by $\mathbf{L}$.

3.5. Well-Formedness

A global type is well-formed if it is a valid description of an ABS system, i.e. its translation into a regular expression only describes well-formed histories and at the end of an execution the system has terminated. In order to verify this, we impose the additional restriction that every method (which can be called several times in a type) must have the same method-local type for each future, that results from a call on this method. The main part to verify a global type, is to check whether the projection on all futures occurring in it is defined.

Not every global type captures global histories. for example, when futures are reused there is no permutation that results in a well-formed history. We do not give an analysis on global types whether a global type is well-formed. Instead we check the restriction while projecting to local types.

Definition 45 (Well-Formed Global-Types)
Let $\mathbf{G}$ be a global type. $\mathbf{G}$ is well-formed if

1. it has the form $\mathbf{0} \xrightarrow{f} \mathbf{p}\!:\!m(\emptyset)\ .\ \mathbf{G}'$ for some $\mathbf{G}', f, \mathbf{p}, m$
2. for every object $\mathbf{q}$ that occurs within $\mathbf{G}$ and every future f that occurs for $\mathbf{q}$ within $\mathbf{G}$ the projection is defined
$$\forall \mathbf{q} \in \mathsf{Ob}.\ \forall f \in \mathsf{Fut}.\ \mathbf{G} \upharpoonright_{\mathbf{q}} \upharpoonright_f \neq \bot$$
3. every branch ends in end
4. the type that results from replacing every end action with skip is self-contained
5. for every method m within $\mathbf{G}$ all calls on m are projected on types which are pairwise future-equivalent
$$\begin{aligned}&\forall m \in \mathsf{Met}.\ \forall \mathbf{p}, \mathbf{q} \in \mathsf{Ob}.\\ &\Big(\exists j, k \in \mathbb{N}.\ \exists f, f' \in \mathsf{Fut}.\\ &\big(\mathsf{action}(j) = \mathbf{p} \xrightarrow{f} \mathbf{q}\!:\!m \wedge \mathsf{action}(k) = \mathbf{p}' \xrightarrow{f'} \mathbf{q}'\!:\!m\big) \rightarrow \mathbf{G} \upharpoonright_q \upharpoonright_f \equiv_{\mathsf{Fut}} \mathbf{G} \upharpoonright_{q'} \upharpoonright_{f'} \Big)\end{aligned}$$
In this case we denote with $\mathbf{G} \upharpoonright_m$ the method-local type of m: $\mathbf{G} \upharpoonright_m = \mathbf{G} \upharpoonright_{\mathbf{q}} \upharpoonright_f$ for the according $\mathbf{q}$ and some f.

Condition 2 ensures that the type describes only valid runs, conditions 3 and 4 ensure that the described communication is fully resolved and condition 5 ensures that every method call describes the same communication. Condition 1 is needed to ensure a clean start, i.e. only one object is active.

Theorem 1 (Type Well-Formedness implies History Well-Formedness)
If a global type $\mathbf{G}$ is well-formed, then each history in $\mathcal{L}(\tau(\mathbf{G}))$ is well-formed.

The proof is given in the appendix.

We do not give a notion of well-formedness of local types, because we are only able to reason about local types if we know the global type it was projected from. We justify this in the next section.

A global type describes a system with a fixed number of objects. We say that a system fits a global type if it has the correct classes and objects, and its objects are initialized.

Definition 46 (Fitting System)
Let $S = (\overrightarrow{Sys}, \overrightarrow{Sch}, h)$ be an ABS system and A the ABS data type system of S. Let $\mathbf{G} = \mathbf{0} \xrightarrow{f} \mathbf{q}\!:\!m(\emptyset)\ .\ \mathbf{G}'$ be a well-formed global type, Ob the set of objects in $\mathbf{G}$ and $methods(\mathbf{q})$ the set of methods called on $\mathbf{p}$ in $\mathbf{G}$.

The system S *fits* $\mathbf{G}$ if the following holds:

1. For each $\mathbf{q} \in Ob$, there is a class $Cl_{\mathbf{q}}$ in A, and $\mathbf{q}$ is the name of an object in $\mathbf{im}(\overrightarrow{Sys})$.
2. For each $\mathbf{q} \in Ob$, there is a field $fl_{\mathbf{q}}$ in every class $Cl_{\mathbf{r}}, \mathbf{r} \neq \mathbf{q}$ with $\sigma(fl_{\mathbf{q}}) = \mathbf{q}$

The first point connects the objects of global types to their counterpart in data types. The second point models that all objects already have pointers to each other.

A system S *initially fits* $\mathbf{G}$ if additionally the following holds:

1. The only active object is $\mathbf{p}$ with the history $[\mathsf{invREv}(\mathbf{0}, \mathbf{P}, f, m, \epsilon))]$ for some expression list e.
2. The global history is $h = [\mathsf{invREv}(\mathbf{0}, \mathbf{P}, f, m, \epsilon))]$
3. Every other history is empty.

The first point models that the system is initialized correctly. The others model that the history is initialized only with the first event.

4. Verification

4.1. Admissibility

Session types allow us to locally check program code to ensure global guarantees. For our system, this means that if every method is captured by its method-local type, then the system is captured by its global type. If a system is captured by its global type $\mathbf{G}$, this especially means that no object can distinguish between the histories in $\tau(\mathbf{G})$ and its well-formed permutations. However the guarantee that all methods are captured by their types, is not strong enough to guarantee this for objects: on object-level the method-local types can be interleaved differently than specified in the global type. No process sees a difference between the actual history and the specified history, but an object does.

A global type is admissible if it additionally guarantees that the processes can be interleaved in only one way: A global type $\mathbf{G}$ is admissible if for every object $\mathbf{p}$ in every history captured by $\mathbf{G}$ all permutations are not distinguishable from the view of $\mathbf{p}$. I.e., the *local* history of $\mathbf{p}$ is the same in every permutation.

To model that a type describes a local behavior independent from the permutations resulting from concurrency and delays, we extend linear types from [18]. We regard two behaviors of an object to be equal, if the order in which the single processes are executed and interleaved are the same.

Formally we define:

Definition 47 (Local-Order Preservation)
Let h be a global history and $\mathbf{p}$ an object. A permutation h' of h is *local-order preserving* for $\mathbf{p}$ if the history of their projection to $\mathbf{p}$ is equal

$$h\restriction_{\mathbf{p}} = h'\restriction_{\mathbf{p}}$$

Example 17
The following history h is not local-order preserving. The history h' is a permutation of h, both h and h' are well-formed, but the local history of $\mathbf{A}$ differs: The events for executing f and f' are swapped.

$$h = \big[\mathsf{invREv}(\mathbf{0}, \mathbf{B}, f_0, m_0, \epsilon), \mathsf{invEv}(\mathbf{B}, \mathbf{A}, f, m, \epsilon), \underbrace{\mathsf{invREv}(\mathbf{B}, \mathbf{A}, f, m, \epsilon), \mathsf{futEv}(\mathbf{A}, f, m, \epsilon)}_{f}\big] \circ$$

$$\big[\mathsf{invEv}(\mathbf{B}, \mathbf{A}, f', m2, \epsilon), \underbrace{\mathsf{invREv}(\mathbf{B}, \mathbf{A}, f', m2, \epsilon), \mathsf{futEv}(\mathbf{A}, f', m2, \epsilon)}_{f'}, \mathsf{futEv}(\mathbf{B}, f_0, m_0, \epsilon)\big]$$

$$h\restriction_{\mathbf{A}} = \big[\mathsf{invREv}(\mathbf{B}, \mathbf{A}, f, m, \epsilon), \mathsf{futEv}(\mathbf{A}, f, m, \epsilon), \mathsf{invREv}(\mathbf{B}, \mathbf{A}, f', m2, \epsilon), \mathsf{futEv}(\mathbf{A}, f', m2, \epsilon)\big]$$

$$h' = \big[\mathsf{invREv}(\mathbf{0}, \mathbf{B}, f_0, m_0, \epsilon), \mathsf{invEv}(\mathbf{B}, \mathbf{A}, f, m, \epsilon), \mathsf{invEv}(\mathbf{B}, \mathbf{A}, f', m2, \epsilon)\big] \circ$$
$$\big[\underbrace{\mathsf{invREv}(\mathbf{B}, \mathbf{A}, f', m2, \epsilon), \mathsf{futEv}(\mathbf{A}, f', m2, \epsilon)}_{f'}, \underbrace{\mathsf{invREv}(\mathbf{B}, \mathbf{A}, f, m, \epsilon), \mathsf{futEv}(\mathbf{A}, f, m, \epsilon)}_{f}, \mathsf{futEv}(\mathbf{B}, f_0, m_0, \epsilon)\big]$$

$$h'\restriction_{\mathbf{A}} = \big[\mathsf{invREv}(\mathbf{B}, \mathbf{A}, f', m2, \epsilon), \mathsf{futEv}(\mathbf{A}, f', m2, \epsilon), \mathsf{invREv}(\mathbf{B}, \mathbf{A}, f, m, \epsilon), \mathsf{futEv}(\mathbf{A}, f, m, \epsilon)\big]$$

A well-formed global type $\mathbf{G}$ is admissible, if every permutation of every described history that is well-formed is also local-order preserving: This means that the object can not distinguish whether the system behaves according to the simple model without asynchronous calls and delays or includes concurrency.

Definition 48 (Admissible Types)
A well-formed global type $\mathbf{G}$ is admissible, if every well-formed permutation of every described history is local-order preserving:

$$\forall h, h' \in \mathsf{Ev}^*. \ \Big(\big(h \in \tau(\mathbf{G}) \land \mathit{wellFormed}(h') \land \pi(h, h')\big) \to \forall \mathbf{p} \in \mathsf{Ob}. \ h\!\restriction_{\mathbf{p}} = h'\!\restriction_{\mathbf{p}}\Big)$$

Example 18
The following type is not admissible, because it describes the history h from Example 17:

$$\mathbf{0} \xrightarrow{f_0} \mathbf{B}\!:\!m_0 \ . \ \mathbf{B} \xrightarrow{f} \mathbf{A}\!:\!m \ . \ \mathbf{A}\!\downarrow\! f \ . \ \mathbf{B} \xrightarrow{f'} \mathbf{A}\!:\!m2 \ . \ \mathbf{A}\!\downarrow\! f' \ . \ \mathbf{B}\!\downarrow\! f_0 \ . \ \mathsf{end}$$

A type with unbounded repetition describes an infinite language - its translation into a regular expression describes an infinite language. It is not possible to check all histories for local-order preservation by computing all their permutations. We show that it suffices to check a derived type describing a finite language to decide admissibility. We use star-height, analogous to star-height of regular expressions, as a measure for types.

Definition 49
The star-height $\mathit{sh}(\mathbf{G})$ of a global type $\mathbf{G}$ is the maximal number of nodes labelled with $*$ on any branch in the AST of $\mathbf{G}$.

Given a global type with unbounded repetitions, we can construct two global types with one unbounded repetition less, by replacing one unbounded repetition once with k-bounded repetition with $k = 0$ and once with $k = 2$.

Checking whether a permutation h' of h is local-order preserving is checking whether there are two events i, j with $i < j$ which are issued by the same objects that are permuted to positions i', j' with $j' < i'$. One must check whether this can be the case for a position i before the repetition and j resulting from inside the repetition. But as the subhistory resulting from the repetition is a sequence of *future-equivalent* copies, it suffices to check whether there is such a j in the first iteration.

Checking only the first repetition (which would correspond to $k = 1$) does cover all cases. Additionally it must be checked that there are no such i, j for *two* arbitrary subsequences resulting from the repetition. But as all such subsequences are future-equivalent, it suffices to check the first and second one. This leads us to bound one repetition with $k = 2$.

Additionally, in the type with $k = 0$ one can check whether two events are swappable if the repetition is not executed. The following example illustrates why this case is necessary:

Example 19
The following type is not admissible for $k = 0$, but admissible for any $k > 0$.

$$\mathbf{0} \xrightarrow{f_0} \mathbf{B}\!:\!m_0 \ . \ \mathbf{B} \xrightarrow{f} \mathbf{A}\!:\!m \ . \ \mathbf{A}\!\downarrow\! f \ . \ \big(\mathbf{B}\!\uparrow\! f\big)^k \ . \ \mathbf{B} \xrightarrow{f'} \mathbf{A}\!:\!m \ . \ \mathbf{A}\!\downarrow\! f' \ . \ \mathbf{B}\!\downarrow\! f_0 \ . \ \mathsf{end}$$

The following type is only admissible for $k = 0$, but not admissible for any $k > 0$.

$$\mathbf{0} \xrightarrow{f_0} \mathbf{A}\!:\!m_0 \ . \ \big(\mathbf{A} \xrightarrow{f} \mathbf{B}\!:\!m \ . \ \mathbf{B}\!\downarrow\! f \ . \ \mathbf{A} \xrightarrow{f'} \mathbf{B}\!:\!m_2 \ . \ \mathbf{B}\!\downarrow\! f'\big)^k \ . \ \mathbf{A}\!\downarrow\! f_0 \ . \ \mathsf{end}$$

This is the only case where the unsynchronized calls on $\mathbf{B}$ are not executed.

The formal description of replacing unbounded repetition:

Definition 50
A type $\mathbf{G}$ is more bounded than $\mathbf{G}'$, $\mathbf{G} \preceq_{0,2} \mathbf{G}'$ if $\mathbf{G}$ is equal to $\mathbf{G}'$ after one unbounded repetition in $\mathbf{G}'$ is replaced by some bounded repetition with $k = 0$ or $k = 2$ or $\mathbf{G} = \mathbf{G}'$. We denote the transitive closure with $\mathbf{G} \preceq^*_{0,2} \mathbf{G}'$.

The unrolling $u_{0,2}(\mathbf{G})$ is the set of all types which are maximally bound, i.e. more bounded than $\mathbf{G}$ and have no further unbounded repetition.

$$u_{0,2}(\mathbf{G}) = \{\mathbf{G}' \mid \mathbf{G}' \preceq^*_{0,2} \mathbf{G} \wedge sh(\mathbf{G}') = 0\}$$

This produces an exponential amount of types, as $|u_{0,2}(\mathbf{G})| \in \mathcal{O}(2^{|\# \text{ of } * \text{ operators in } \mathbf{G}|})$. But all of them describe finite languages.

Lemma 1 (Decidability of Admissibility)
A well-formed global type $\mathbf{G}$ is not admissible iff some type in $u_{0,2}(\mathbf{G})$ is not admissible.

The proof is in the appendix.

When reasoning about branching types, it is easier to reason about the branches alone.

Definition 51 (Linear Types)
A global type $\mathbf{G}$ or a local type $\mathbf{L}$ is linear if all branching operators within are repetitions.

There are $n!$ permutations to check for a history of length n. Instead of computing all permutations and checking whether they are well-formed and local order-preserving, we extend the analysis of [18] and use causality graphs to check a linear type in polynomial time for admissibility.

Definition 52 (Causality Graph)
Let $h \in Ev^*$ be a history, its causality graph $\mathfrak{C}(h) = (V, E)$ is defined as follows. All free variables are implicitly quantified by an existential quantifier with the scope of the surrounding conjunct.

$$\begin{aligned}
& V = \big\{1, \dots, |h|\big\}, && (4.1)\\
\forall i, j \in V.\ E(i,j) \iff\ & h[i] = \mathsf{invEv}(\mathbf{p}, \mathbf{q}, f, m, e) \wedge h[j] = \mathsf{invREv}(\mathbf{p}, \mathbf{q}, f, m, e) \vee && (4.2)\\
& h[i] = \mathsf{futEv}(\mathbf{p}, f, m, \epsilon) \wedge h[j] = \mathsf{futREv}(\mathbf{q}, f, \epsilon) \vee && (4.3)\\
& h[i] = \mathsf{throwEv}(\mathbf{p}, f, m, \epsilon) \wedge h[j] = \mathsf{throwREv}(\mathbf{q}, f, \epsilon) \vee && (4.4)\\
& \big(\exists \mathbf{p} \in \mathsf{Ob}.\ h[i] \upharpoonright_{\mathbf{p}} \neq \epsilon \wedge h[j] \upharpoonright_{\mathbf{p}} \neq \epsilon \wedge act(i, \mathbf{p}) = act(j, \mathbf{p})\big) \vee && (4.5)\\
& \exists f, f', f'' \in \mathsf{Fut}.\ \exists \mathbf{p}, \mathbf{q} \in \mathsf{Ob}.\ \mathbf{p} \neq \mathbf{q} \wedge act(i, \mathbf{p}) = f \wedge h[j] \upharpoonright_{\mathbf{p}} \neq \epsilon \wedge && (4.6)\\
& \quad last(h[1..j-1] \upharpoonright_{\mathbf{q}}) = \mathsf{awaitEv}(\mathbf{q}, f, f'') \wedge && (4.7)\\
& \quad \big(h[i] = \mathsf{futEv}(\mathbf{p}, f, m, \epsilon) \vee h[i] = \mathsf{throwEv}(\mathbf{p}, f, m, \epsilon)\big) && (4.8)
\end{aligned}$$

The disjuncts in (4.2), (4.3) and (4.4) link events according to the well-formedness of histories. The disjunct in (4.5) links events from the same active future. The disjunct in (4.8) links resolving events to the first events from the future they reactivate.

The causality graph $\mathfrak{C}(\mathbf{G})$ is the disjoint union of the causality graphs of all histories for all types in its unrolling. Let $H(\mathbf{G}) = \{h \mid h \in \tau(\mathbf{G}) \wedge \mathbf{G} \in u_{0,2}(\mathbf{G})\}$

$$\mathfrak{C}(\mathbf{G}) = \{ \dot{\bigcup_{\substack{(V,E)=\mathfrak{C}(h)\\ h\in H(\mathbf{G})}}} V, \dot{\bigcup_{\substack{(V,E)=\mathfrak{C}(h)\\ h\in H(\mathbf{G})}}} E\}$$

The causality graph encodes causality; if there is a path between two nodes then either the first causes the second (e.g., the first is sending and the second is receiving a call) or they happen in this order in a single process and the first thus must be executed first.

Lemma 2 (Causality)
Let $\mathbf{G}$ be a well-formed linear global type which contains no unbounded repetition and $\mathfrak{C}(\mathbf{G})$ its causality graph. Let $I(\mathbf{p})$ be the set of indices, where $\tau(\mathbf{G})[i]\upharpoonright_{\mathbf{p}} \neq \epsilon$ for an object $\mathbf{p}$.

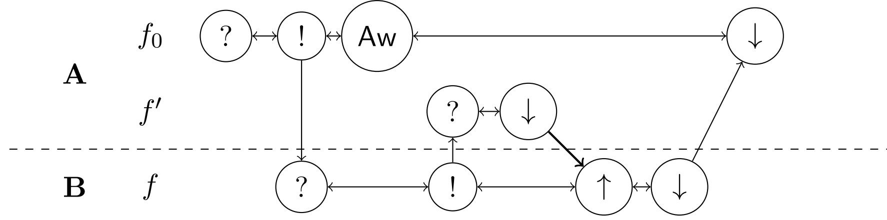
A well-formed global type $\mathbf{G}$ is admissible iff for each $\mathbf{p} \in \mathsf{Ob}$ for each $i, j \in I(\mathbf{p}), i < j$, where i, j are in the same connected component and $i < j$, there is a path from i to j in $\mathfrak{C}(\mathbf{G}) = (V_{\mathbf{G}}, E_{\mathbf{G}})$.

The proof is in the appendix. A single-source-all-paths search in a DAG has $\mathcal{O}(|V| + |E|)$ computation time [11]. There are at most $|V|$ elements of $I(\mathbf{p})$ for which such a search has to be executed. Thus we can bound for one history the computation time with $\mathcal{O}(n^2)$. However there are possibly exponentially many histories in $u_{0,2}(\mathbf{G})$.

Example 20
Consider the following global type

$$\mathsf{U} = \mathbf{0} \xrightarrow{f_0} \mathbf{A}{:}m() \;.\; \mathbf{A} \xrightarrow{f} \mathbf{B}{:}m2() \;.\; \mathsf{Rel}(\mathbf{A}, f) \;.\; \mathbf{B} \xrightarrow{f'} \mathbf{A}{:}m() \;.\; \mathbf{A}{\downarrow}f' \;.\; \mathbf{B}{\uparrow}f' \;.\; \mathbf{B}{\downarrow}f \;.\; \mathbf{A}{\downarrow}f_0$$

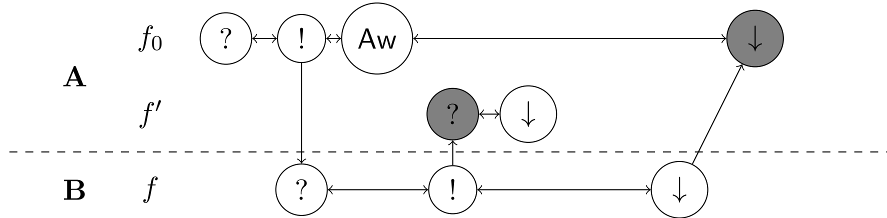
Its causality graph is



The nodes are line-wise aligned by the active future. E.g. the Aw node results from the $\mathsf{awaitEv}$ in the translation of U. It is easy to see that all events for $\mathbf{A}$ are pair-wise connected. Now consider the following global type, where $\mathbf{B}$ does not read from f'

$$\mathsf{U}' = \mathbf{0} \xrightarrow{f_0} \mathbf{A}{:}m() \;.\; \mathbf{A} \xrightarrow{f} \mathbf{B}{:}m2() \;.\; \mathsf{Rel}(\mathbf{A}, f) \;.\; \mathbf{B} \xrightarrow{f'} \mathbf{A}{:}m() \;.\; \mathbf{A}{\downarrow}f' \;.\; \mathbf{B}{\downarrow}f \;.\; \mathbf{A}{\downarrow}f_0$$

Its causality graph is



Now there is no path between the two gray nodes, thus the type is not admissible.

The type is not admissible, because the single history in $\mathcal{L}(\tau(\mathsf{U}'))$ has a permutation which is well-formed *but has swapped the order* of the events corresponding to the gray nodes: This is because the computation of f' can start *after* the computation of f_0 terminates. In the first example this is not possible, because f_0 is reactivated after f' has been computed, now the invocation reaction f' can be delayed and starts after f_0 has been reactivated.

4.2. Translation of Method-Local Types into ABSDL

To verify the method-local types, we use the symbolic execution of KeY to generate the set of *symbolic* histories which a method can realize. We construct a formula that holds if a symbolic history is described by a method-local type. It is not possible to use established translations of regular expressions into formulas [6], because

- *symbolic* histories describe a *set* of histories and
- the type describes the history of a process, but the symbolic execution describes the history of the *object*.

We deal with the first point by bounding elements of the events with existential quantifiers. These can be instantiated with terms to match the generated symbolic history. The second point is dealt with by marking the events of the history which are not issued by the process. 4 We translate linear repetition-free types to terms and combine the translations of branching and repeating types with disjunctions and the *pattern* predicate. The objects in the events are mapped to *self* and fields, the futures are mapped to variables and the passed parameters and return values are described by additional formulas.

The translation of linear repetition-free types is straightforward as every action maps to one event, and every event maps to one function symbol.

Definition 53 (Translation of Linear Repetition-Free Types)
Let $\mathbf{L}$ be a linear repetition-free method-local type for a method m in $\mathbf{p}$. The translation $T(\mathbf{L})$ of $\mathbf{L}$ is a term of type Seq and a formula describing how futures are passed. $C(\mathbf{Exception})$ is the set of constructors of $\mathbf{Exception}$. The predicated $issuedBy, outer$ and $paramAt$ are introduced in Section 2.2: $issuedBy(s,f)$ holds is all events in the sequence s are issued by the process computing f, $outer(v,C)$ holds if the outermost constructor of the value v is C and $paramAt(v,f,i)$ holds if f the ith parameter of v.

$$
\begin{aligned}
T &: \mathsf{LT} \to \mathsf{Trm} \times \mathsf{Frm}\\
T(\mathbf{L}) &= (t_\mathbf{L}, \psi_\mathbf{L})\\
T(\mathbf{q}!_f m(R)) &= \big([\mathsf{invEv}(\mathit{self}, \mathit{self}.\mathit{fl}_\mathbf{q}, f, m, v)], \bigwedge_{(i,f')\in R} v[i] = f'\big)\\
T(\mathbf{q}?_f m(R)) &= \big([\mathsf{invREv}(q, \mathit{self}, f, m, v)], \bigwedge_{(i,f')\in R} v[i] = f'\big)
\end{aligned}
$$

$$
T(\mathsf{Put}\ f : (C,R)) = \begin{cases} \big([\mathsf{futEv}(\mathit{self}, f, m, v)], outer(v,C) \land \bigwedge_{(i,f')\in R} paramAt(v,f',i)\big) & \text{if } C \notin C(\mathbf{Exception})\\ \big([\mathsf{throwEv}(\mathit{self}, f, m, v)], outer(v,C) \land \bigwedge_{(i,f')\in R} paramAt(v,f',i)\big) & \text{if } C \in C(\mathbf{Exception}) \end{cases}
$$

$$
T(\mathsf{Get}\ f : (C,R)) = \begin{cases} \big([\mathsf{futREv}(\mathit{self}, f, v)], outer(v,C) \land \bigwedge_{(i,f')\in R} paramAt(v,f',i)\big) & \text{if } C \notin C(\mathbf{Exception})\\ \big([\mathsf{throwREv}(\mathit{self}, f, v)], outer(v,C) \land \bigwedge_{(i,f')\in R} paramAt(v,f',i)\big) & \text{if } C \in C(\mathbf{Exception}) \end{cases}
$$

$$
\begin{aligned}
T(\mathsf{Await}(f,f')) &= \big([\mathsf{awaitEv}(\mathit{self}, f, f')] \circ s, \neg issuedBy(s,f)\big)\\
T(\mathbf{L}_1\ .\ \mathbf{L}_2) &= \big(t_{\mathbf{L}_1} \circ t_{\mathbf{L}_2}, \psi_{\mathbf{L}_1} \land \psi_{\mathbf{L}_2}\big)
\end{aligned}
$$

where v, s, q are variables of $Any, Seq\langle\mathsf{Ev}\rangle$ and Ob type. We assume that every variable that is introduced in this way is unique in the translation: when computing $T(\mathbf{L}_1\ .\ \mathbf{L}_2)$, if such a variable occurs in $t_{\mathbf{L}_1}$ and $t_{\mathbf{L}_2}$, we assume it is renamed consistently in $t_{\mathbf{L}_2}$.

The constants for method names m in $T(\mathsf{Put}\ f:(C,R))$ and $T(\mathsf{Get}\ f:(C,R))$ are taken from the corresponding $\mathbf{p}?_f m(R)$ (resp. $\mathbf{p}!_f m(R)$) event.

The additional sequence and the formula of $T(\mathsf{Await}(f,f'))$ are necessary because the symbolic execution keeps track of the history on object-level. Thus after executing an Await statement, a sequence is appended to *history*. To incorporate this sequence, a variable is added after the Await event and marked as not issued by this process. Otherwise the sequence could be instantiated with events from the process in question.

Example 21
Consider the following type

$$\mathbf{L} = \mathbf{A}?_{f_0}:m()\ .\ \mathbf{B}!_f:m_2()\ .\ \mathsf{Await}(f_0,f)\ .\ \mathsf{Get}\ f:C_1\ .\ \mathsf{Put}\ f_0:C_2(\{(1,f)\})$$

$$\begin{aligned} T(\mathbf{L}) = \Big(&\big[\mathsf{invREv}(A, \mathit{self}, f_0, m, ex), \mathsf{invEv}(\mathit{self}, \mathit{self}.\mathit{fl}_{\mathbf{B}}, f, m_2, ex_1), \mathsf{awaitEv}(\mathit{self}, f_0, f)\big] \circ s \circ \\ &\big[\mathsf{futREv}(\mathit{self}, f, ex_2), \mathsf{futEv}(self, f_0, m, ex_3)\big], \\ &\neg \mathit{issuedBy}(s, f_0) \land \mathit{outer}(ex_2, C_1) \land \mathit{outer}(ex_3, C_2) \land \mathit{paramAt}(ex_3, f, 1)\Big) \end{aligned}$$

The term of the translation contains several free variables: the parameter lists, the futures, the event sequence during the suspension and the caller object. These variables are bound by existential quantifiers in the final invariant.

To deal with repetition and branching inside repetitions we disassemble the type into linear segments by replacing repetitions by variables and splitting branches into linear types. First we extend local types with variables.

Definition 54 (Local Types with Variables)
Let V be a set of variable symbols. The set of *local types with variables* LT^+ is a superset of the set of local types, defined by the following extension of the grammar for local types:

$$\mathbf{L} ::= \dots \ \big|\ t$$

The translation of a type variable t is a logical variable s_t of sequence type.

$$T(t) = (s_t, \mathbf{true})$$

Intuitively a labelled local type describes an equality, i.e. that t can be replaced by $\mathbf{L}$. Type variables allow us to formulate the relation between branches by formulas.

To disassemble a type, we use two operations: *Flattening* removes one repetition from a linear type and replaces it with a type variable. *Debranching* splits a non-linear type into branches.

Definition 55 (Debranching)
Let $(t,\mathbf{L})$ be a labelled non-linear method-local type. The debranching of $\mathbf{L}$ is the set of all linear types obtained by splitting the first branching in $\mathbf{L}$. As $\mathbf{L}$ is non-linear, it has the form $\mathbf{L}'.\&_f\{\mathbf{L}_j\}_{j\in J}$ or $\mathbf{L}'.\oplus\{\mathbf{L}_j\}_{j\in J}$. We assume that J is an interval in $\mathbb{N}$: $J = \{1,2,\dots,n\}, n > 1$. Then the debranching of $(t,\mathbf{L})$ is

$$\begin{aligned} \mathfrak{D} : V \times \mathsf{LT}^+ \times \mathbb{N} &\to \big(V \times \mathcal{P}(V)\big) \times \mathcal{P}(V \times \mathsf{LT}^+) \\ \mathfrak{D}(t,\mathbf{L},i) &= \big((t, \{t_i,\dots,t_{i+|J|-1}\}), \{(t_{i+k-1}, \mathbf{L}'\ .\ \mathbf{L}_k)\}_{k\in J}\big) \end{aligned}$$

The second component of the result is the set of all branches, labelled with a fresh type variable. The first component is the original variable and the set of newly introduced variables. The i parameter is needed to ensure that every type variable identifies a type.

Example 22
Consider the following type

$$(t_0, \mathbf{L}) = \mathbf{A}?_f m(\{(1, f')\}) \;.\; \&_{f'} \left\{ \begin{array}{l} \mathsf{Get}\ f' : ok \;.\; \mathsf{Put}\ f \\ \mathsf{Get}\ f' : deny \;.\; (\mathbf{B}!_{f''} m_1 \;.\; \mathsf{Get}\ f'')^* \;.\; \mathsf{Put}\ f \\ \mathsf{Get}\ f' : \mathbf{PatternFailure} \;.\; \oplus \left\{ \begin{array}{l} \mathsf{Put}\ f : C_1 \\ \mathsf{Put}\ f : C_2 \end{array} \right\} \end{array} \right\}$$

Its debranching is

$$\begin{aligned} \mathfrak{D}(t_0, \mathbf{L}, 1) = \Big(& (t_0, \{t_1, t_2, t_3\}), \\ & \{ (t_1, \mathbf{A}?_f m(\{(1, f')\}) \;.\; \mathsf{Get}\ f' : ok \;.\; \mathsf{Put}\ f), \\ & \quad (t_2, \mathbf{A}?_f m(\{(1, f')\}) \;.\; \mathsf{Get}\ f' : deny \;.\; (\mathbf{B}!_{f''} m_1 \;.\; \mathsf{Get}\ f'')^* \;.\; \mathsf{Put}\ f), \\ & \quad (t_3, \mathbf{A}?_f m(\{(1, f')\}) \;.\; \mathsf{Get}\ f' : \mathbf{PatternFailure} \;.\; \oplus \left\{ \begin{array}{l} \mathsf{Put}\ f : C_1 \\ \mathsf{Put}\ f : C_2 \end{array} \right\}) \\ & \} \Big) \end{aligned}$$

We use the following operation to replace repetitions in a local type and receive a set of labelled local types which describes the same set of histories.

Definition 56 (Flattening)
Let $n \in \mathbb{N}$ be a number and $(t, \mathbf{L})$ be a method local type such that

$$\mathbf{L} = \mathbf{L}_0 \;.\; (\mathbf{L}'_0)^* \;.\; \mathbf{L}_1 \;.\; (\mathbf{L}'_1)^* \;.\; \dots \;.\; (\mathbf{L}'_{n-1})^* \;.\; \mathbf{L}_n$$

Where all $\mathbf{L}_i$ are repetition free. Let $(t, \mathbf{L})$ be a method local type and $n \in \mathbb{N}$ the number of repetitions operators in $\mathbf{L}$. The *flattening* $\mathfrak{F}(t, \mathbf{L}, i)$ is a set of labelled local types and a set of type variables.

$$\begin{aligned} \mathfrak{F} :& V \times \mathsf{LT}^+ \times \mathbb{N} \to \mathcal{P}(V) \times \mathcal{P}(V \times \mathsf{LT}^+) \\ \mathfrak{F}(t, \mathbf{L}, i) =& \{(t_i, \mathbf{L}_0 \;.\; t_{i+1} \;.\; \mathbf{L}_1 \;.\; t_{i+2} \dots \;.\; t_{i+n-1} \;.\; \mathbf{L}_n), (t_{i+k}, \mathbf{L}'_k)_{k<n}\}, \{t_{i+1}, \dots, t_{i+n-}\} \end{aligned}$$

The first component is the set of introduced variables. The second component is the set of repeated segments, labelled with new type variables and the input type with removed variables.

We apply these operation recursively to obtain

- A set of labelled linear repetition free types
- A set of type variables marked for repetition
- An assignment from variables to sets of variables, which describes inner debranchings

The algorithm manages a set W of labelled types, which still must be disassembled. Any of these types is picked and if it is not linear, the type is split into branches. The branches are labelled with new variables and the B function maps the original variable to the set of the

variables for the branches. If the type is linear but not repetition free, it is flattened. The removed parts are added to W and their variables are added to R. If the type is linear and repetition free it is added to the result mapping T.

In the end, T maps all occuring variables to linear repetition-free segments and R contains all variables whose type is repeated. B maps all variables to the set of variables which label types resulting from debranching.

Definition 57 (Disassembly of Method-Local types)
Let $\mathbf{L}$ be a linear method-local type for a method m. The disassembly of $\mathbf{L}$ is computed by the following algorithm

input linear method-local type $(t, \mathbf{L})$
output $T : V \rightharpoonup \mathsf{LT}^+, R \in \mathcal{P}(V), B : V \rightharpoonup \mathcal{P}(V)$
$W = \{(t_0, \mathbf{L})\}$\\set of possibly non-linear types
$R = \emptyset$\\set of variables, whose type is inside a repetition
$\forall v.\ B(v) = \bot$\\maps variables to the branches of their type
$\forall v.\ T(v) = \bot$\\maps variables to linear repetition-free types
$i = 1$
while$(W \neq \emptyset)$**do**
  pick any $(t, \mathbf{L}') \in W$
  $W = W \setminus \{(t, \mathbf{L}')\}$\\remove it from working set
  if $\mathbf{L}'$ is not linear
    $(T', B') = \mathfrak{D}(t, \mathbf{L}', i)$\\debranch if not linear
    $B = B[t \mapsto B']$
    $W = W \cup T'$
    $i = i + |T'|$
  elseif $\mathbf{L}'$ is not repetition-free
    $(T', R') = \mathfrak{F}(t, \mathbf{L}', i)$\\flatten if not repetition-free
    $R = R \cup R'$
    $W = W \cup T'$
    $i = i + |T'|$
  else
    $T = T[t \mapsto \mathbf{L}']$\\add to result
od
return (T, R, B)

We write $\mathfrak{T}(\mathbf{L}) = (T, R, B)$ for the result of the algorithm.

Example 23
Consider the type

$$\mathbf{A}?_f m\ .\ \left(\& \left\{ \begin{array}{l} \mathbf{C}!_{f''} m_3\ .\ (\mathsf{Get}\ f'')^* \\ \mathbf{B}!_{f'} m_2 \end{array} \right\} \right)^*\ .\ \mathsf{Put}\ f$$

Its disassembly $\mathfrak{T}(\mathbf{L})$ is

$$\begin{aligned} T = \{&(t_0, \mathbf{A}?_f m \ . \ t_1 \ . \ \mathsf{Put}\ f), \\ &(t_2, \mathbf{C}!_{f''} m_3 \ . \ t_4), \\ &(t_3, \mathbf{B}!_{f'} m_2), \\ &(t_4, \mathsf{Get}\ f'')\} \\ R = \{&t_1, t_4\} \\ B = \{&(t_1, \{t_2, t_3\})\} \end{aligned}$$

Note that the original type is labelled with t_0 in the initialization of the algorithm and the outermost type is still labelled with t_0.

With the disassembly we can construct a formula, that holds if a sequence of events is captured by a type, if all events that are not issued by this process are removed.

Definition 58 (Translation)
Let $(i_t)_{t \in V}$ be a family of logical variables of **Int** type. Let $\mathbf{L}$ be a linear method-local type and $\mathfrak{T}(\mathbf{L}) = (T, R, B)$ its disassembly. Its translation is

$$\begin{aligned} \chi_{\mathbf{L}} = &\bigwedge_{\substack{t \in R \\ T(t) \neq \bot}} \big(\mathit{pattern}(t, t_{T(t)}, i_t) \wedge \psi_{T(t)}\big) \\ \wedge &\bigwedge_{\substack{t \in R \\ T(t) = \bot}} \big(\mathit{pattern}(t, B(t), i_t)\big) \\ \wedge &\bigwedge_{\substack{t \in \mathbf{dom}(T) \\ t \notin R}} \big(t = t_{T(t)} \wedge \psi_{T(t)}\big) \end{aligned}$$

The first disjunct encodes the repetition marked in R, if the repeated type variable labels a linear repetition-free type. This is the case if a repetition contains no branching. The second disjunct encodes the repetition marked in R, if the repeated type variable t describes a branch. The branches are recorded in $B(t)$. The last disjunct encodes the remaining types.

We quantify over all free variables but t_0, additionally we demand that all futures are not equal to each other.

Definition 59
Let $\mathbf{L}$ be a linear method-local type and $\chi_{\mathbf{L}}$ its ABSDL translation. Let $\{T_1, \dots, T_m\}$ be the set of types such there is at least one free variable of type $\mathsf{Fut}\langle T \rangle$ in $\chi_{\mathbf{L}}$ and let $\{T'_1, \dots, T'_k\} \setminus \{t_0\}$ be the set of types T such there is at least one free variable of type T in $\Psi_{\mathbf{L}}$ and T has not the form $\mathsf{Fut}\langle T' \rangle$.

Let $F_T = \{f_1, \dots, f_n\}$ be the set of free logical variables of type $\mathsf{Fut}\langle T \rangle$ and $V_T = \{v_1, \dots, v_m\}$ the set of free variable of other type T where T has not the form $\mathsf{Fut}\langle T' \rangle$.

$$\begin{aligned} &\varphi_{\mathbf{L}}(t_0) = \\ &\quad \exists f_1^1, \dots, f_n^1 \in \mathsf{Fut}\langle T_1 \rangle. \ \dots \ \exists f_1^m, \dots, f_n^m \in \mathsf{Fut}\langle T_m \rangle. \\ &\quad \exists v_1^1, \dots, v_n^1 \in T'_1. \ \dots \ \exists v_1^k, \dots, v_n^k \in T'_k. \wedge \\ &\quad \chi_{\mathbf{L}} \wedge \bigwedge_{\substack{f, f' \in \bigcup_{T \in T_F} F_T \\ f \neq f'}} f \neq f' \end{aligned}$$

The translation of a non-linear method-local type $\mathbf{L}$ with debranching $(T, B) = \mathfrak{D}(t_0, \mathbf{L}, 1)$ is

$$\varphi_{\mathbf{L}}(t_0) = \bigvee_{\mathbf{L}' \in T} \varphi_{\mathbf{L}'}(t_0)$$

Example 24
Consider the type from example 23

$$\mathbf{L} = \mathbf{A}?_f m \ . \ \left(\& \left\{ \begin{array}{l} \mathbf{C}!_{f''} m_3 \ . \ (\mathsf{Get}\ f'')^* \\ \mathbf{B}!_{f'} m_2 \end{array} \right\} \right) \ . \ \mathsf{Put}\ f$$

Its disassembly $\mathfrak{T}(\mathbf{L})$ is

$$\begin{aligned} T = \{&(t_0, \mathbf{A}?_f m \ . \ t_1 \ . \ \mathsf{Put}\ f), \\ &(t_2, \mathbf{C}!_{f''} m_3 \ . \ t_4), \\ &(t_3, \mathbf{B}!_{f'} m_2), \\ &(t_4, \mathsf{Get}\ f'')\} \\ R = \{&t_1, t_4\} \\ B = \{&(t_1, \{t_2, t_3\})\} \end{aligned}$$

Its translation is

$$\begin{aligned} \varphi_{\mathbf{L}} = &\exists i_1, i_4 \in \mathsf{Int}.\ \exists f, f_1, f_2 \in \mathsf{Fut}.\ \exists t_1, t_2, t_3, t_4 \in \mathsf{Seq}.\ \exists e_1, e_2, e_3, e_4, e_5 \in \mathsf{Any}. \\ & f \neq f_1 \wedge f_1 \neq f_2 \wedge f_1 \neq f_2 \wedge \\ & t_0 = \left[\mathsf{invREv}(A, \mathit{self}, f, m, e_1)\right] \circ t_1 \circ \left[\mathsf{futEv}(\mathit{self}, f, m, e_2)\right] \wedge \\ & t_2 = \left[\mathsf{invEv}(\mathit{self}, \mathit{self}.\mathit{fl}_{\mathbf{B}}, f_1, m_2, e_3)\right] \circ t_4 \wedge \\ & t_3 = \left[\mathsf{invEv}(\mathit{self}, \mathit{self}.\mathit{fl}_{\mathbf{C}}, f_2, m_3, e_4)\right] \wedge \\ & \mathit{pattern}(t_1, \{t_2, t_3\}, i_1) \wedge \mathit{pattern}(t_4, \{\left[\mathsf{futREv}(\mathit{self}, f_2, e_5)\right]\}, i_4) \end{aligned}$$

Let $\mathbf{L}$ be the method-local type for some method m. The formula $\varphi_{\mathbf{L}}(t)$ describes the sequence for the variable t. However, an invariant must hold after the execution of *every* method. Thus we define the following formula as an invariant:

$$\Phi_{\mathbf{L}}(t) = \mathit{lastMethod}(t, m) \rightarrow \varphi_{\mathbf{L}}(t)$$

Where $\mathit{lastMethod}(t, m)$ encodes that the last event was a resolving or throwing action:

$$\begin{aligned} &\mathit{lastMethod}(t, m) = \\ &\quad \exists \mathit{Any}\ e;\ \exists \mathit{Fut}\langle \mathit{Any} \rangle\ f; \\ &\quad\quad t[\mathit{seqLength}(t) - 1] \doteq \mathsf{futEv}(\mathit{self}, m, f, e) \vee t[\mathit{seqLength}(t) - 1] \doteq \mathsf{throwEv}(\mathit{self}, m, f, e) \end{aligned}$$

The formula $\Phi_{\mathbf{L}}(t)$ evaluates to true in every branch of every proof that is an invariant, but the ones for m.

The KeY-ABS prover requires additional information from the global type to produce a proof for that $\Phi_{\mathbf{L}}$ is an invariant. The reason is that we can derive further information about a method-local type from the global type: When reading from a future, the global type guarantees what constructors can by communicated on top-level and whether an exception can be

thrown or not. This can not be encoded into the logic and the proof splits *for every possible outermost constructor*.

Thus we extend the original type with branches for every possible constructor. These branches are needed to close the proof tree branches which are not possible when the global type is known and every other party follows its part of the protocol.

Definition 60
Let $\mathbf{L}$ be a method-local type. The F function maps futures to a set of constructors. $C \in F(f)$ holds if the constructor C is used as a label for f in $\mathbf{L}$.

$$F(f) = \{C \mid \mathsf{futREv}(\mathbf{p}, f, m, C(e)) \in \tau(\mathbf{L}) \lor \mathsf{throwREv}(O, f, m, C(e)) \in \tau(\mathbf{L}) \text{ for some } \mathbf{p}, m, e\}$$

Now $C(f)$ is the set of unused constructors:

$$C(f) = \big(C(\mathsf{Exception}) \cup C(ret(S_{\mathbf{L}}(f)))\big) \setminus F(f)$$

The constructor-complete type $\widetilde{\mathbf{L}}$ results from adding dummy branches for unused constructors of each read future. First every type of the form $\mathsf{Get}\ f\!:\!C\ .\ \mathbf{L}'$ that occurs in $\mathbf{L}$, such that f does not appear as the parameter of an offering in $\mathbf{L}$ is replaced by

$$\&_f\{\mathsf{Get}\ f\!:\!C\ .\ \mathbf{L}'\}$$

Then every offering is completed, i.e., a branch consisting of a reading action and a fresh variable is added for each unused constructor. Every type of the form

$$\&_f\{\mathbf{G}_j\}_{j\in J}$$

is extended with a new index set $K \supseteq J$ and fresh logical variables $t_k, k \in K$ of $Seq\langle\mathsf{Ev}\rangle$ type to $\&_f\{\mathbf{G}_k\}_{k\in K}$, where

$$\mathbf{G}_k = \begin{cases} \mathbf{G}_j & \text{if } k \in J \\ \mathsf{Get}\ f\!:\!C\ .\ t_k & \text{if } C \in C(f) \end{cases}$$

The result is denoted $\widetilde{\mathbf{L}}$.

Example 25
Consider the method-local type

$$\mathbf{L} = \mathbf{A}?_f m\{(1, f')\}\ .\ \mathsf{Get}\ f'\!:\!ok\ .\ \mathsf{Put}\ f$$

In an ABS data type system where

$$C(ret(S_{\mathbf{L}}(f))) = \{ok, deny\}$$
$$C(\mathsf{Exception}) = \{\mathbf{PatternFailure}\}$$

Then

$$\hat{\mathbf{L}} = \mathbf{A}?_f m\{(1, f')\}\ .\ \&'_f \left\{ \begin{array}{l} \mathsf{Get}\ f'\!:\!ok\ .\ \mathsf{Put}\ f \\ \mathsf{Get}\ f'\!:\!deny\ .\ t' \\ \mathsf{Get}\ f'\!:\!\mathbf{PatternFailure}\ .\ t'' \end{array} \right\}$$

Note that the type variables are translated into free sequence variables in $\varphi_{\hat{\mathbf{L}}}$, i.e. the proof closes independently of what m executes after entering such a branch.

Finally we can give the main theorem, that if all methods are captured by their method-local type, then the system is captured by the global type. Recall from Definition 46 that a system initially fits if it has the classes and methods used in $\mathbf{G}$, is initialized and every object has a pointer on every other object.

Theorem 2 (Fidelity)
Let S_0 be an ABS system and $\mathbf{G}$ an admissable global type, such that S_0 initially *fits* to $\mathbf{G}$. If for every method m of S_0 there is a proof that $\Phi_{\widetilde{G\upharpoonright_m}}$ is an invariant, then S_0 is captured by $\mathbf{G}$.

We do not give a guarantee what happens if an object receives calls in the wrong order. Languages or system with typestate[28] guarantee that the specified order is enforced on callee side, but ABS does not offer such a mechanism and we refrain from generating boilerplate code for the ABS classes.

The KeY-ABS prover is only able to deal with partial correctness: It can only guarantee that if a methods terminates then it is captured by its method-local type. It is not able to show that it will always terminate. In this setting, we guarantee deadlock freedom.

Corollary 1 (Dead-Lock-Freedom)
Let S be a system and $\mathbf{G}$ a global type. If S is captured by $\mathbf{G}$ and every process always terminates, then S is deadlock-free.

Proof. Given Theorem 2, it suffices to show that at no node i of the AST of $\mathbf{G}$ a dead-locked state can be described. If there would be such a reachable state, at the corresponding node i of the AST of $\mathbf{G}$, for all objects $\mathbf{p}$ it would hold that $act(i, \mathbf{p}) = \bot$ but there is some f with $wait(i, f) \neq \emptyset$. If this is the last action end, $\mathbf{G}$ is not well-formed because the projection of end requires for any object that $\forall f \in \mathsf{Fut}.\ wait(i, f) = \emptyset$. If it is not the case, then the next action can not be projected because for every action for some object it is required that $act(i, \mathbf{p}) \neq \bot$. The projection for this object would be not defined, thus $\mathbf{G}$ is not well-formed. □

4.3. Scheduling with Session Automata

The admissibility check we introduce in Section 4.1 ensures that the system behaves according to the global type for *every* possible scheduler. This demands that the communication pattern itself enforces a the order of process activations and reactivations.

An alternative is to construct a constraint on the scheduler from the object type – if the scheduler of each object behaves according to this constraint, then Theorem 2 holds also for non-admissible but well-formed global types. We use *session automata* to represent scheduler policies and object types. Session automata are finite state automata which work on words over *infinite alphabets*, are equipped with a finite memory to store read values and can take the values stored in the memory into account when firing a transition.

When the object is idle, the object's scheduler inputs the processes which can be (re-) activated to the session automaton. If a transition labeled with the corresponding (re-) activation event can fire in the automaton, the object (re-) activates this process. If there are several processes which can run such a transition, the scheduler randomly selects one of these processes. This mechanism is a variant of typestate [28]: the scheduler actively guides the object instead of simply observing the order of method calls.

It does not suffice to consider classical NFAs with the alphabet of all method names.

Example 26
Consider the following type

$$\mathbf{p}?_f m \ . \ \mathsf{Rel}(f, f'') \ . \ \mathbf{p}?_{f'} m \ . \ \mathsf{Rel}(f', f'') \ . \ \mathsf{React}(f) \ . \ \mathbf{L}$$

Both processes started are computing the same method and are waiting for the same future. To schedule the correct reactivation (i.e. the first one) the scheduler must keep track of the futures.

Session automata [5] are a subclass of register automata [22]; particularly, they require the freshness of symbols copied into the store. We only store futures when reading a process activation, so session automata satisfies our needs. We define k-register session automata as follows:

Definition 61 (k-Register Session Automata)
Let Σ be a finite set of labels, D be an infinite set of data equipped with equality, and $k \in \mathbb{N}$. A *k-Register Session Automaton* is a tuple (Q, q_0, Δ, F) where

- Q is the finite set of states,
- $q_0 \in Q$ is the initial state,
- $\Delta \subseteq \Big(Q \times Q'\Big) \cup \Big(Q \times \Sigma \times 2^{\{1,\ldots,k\}} \times \{1, \ldots, k\} \times Q\Big)$ is the transition relation,
- $F \subseteq Q$ is the set of accepting states.

Data words are words over an alphabet $\Sigma \times D$. A data word automata has a data store, which can save k data values. A transition fires for a letter $(a, d) \in \Sigma \times D$ if a set of equalities of the form $d = r_i$ are satisfied, where r_i refers to the ith stored data value. After a transition fires, d will be stored in the data store.

The store has one register for each future in the type. While there are only finitely many futures in the type, the futures inside repetitions are only *placeholders*: they correspond to multiple futures in the history. Let $\mathit{fpos}_{\mathbf{L}} : \mathsf{Fut} \to \mathbb{N}$ be an injective function which maps every future in the type $\mathbf{L}$ to a number, such that $\mathbf{im}(\mathit{fpos}_{\mathbf{L}})$ is an interval $(1..k)$ in $\mathbb{N}$. The upon reading a data value with future f, it is stored in $\mathit{fpos}_{\mathbf{L}}(f)$. When the future must be reactivated, the automaton can look up $\mathit{fpos}_{\mathbf{L}}(f)$ to see whether it is the correct one. As all repetition are self-contained, it is safe to overwrite futures which are used only inside a repetition in each iteration - it is guaranteed that they may not be used again.

Let $\sigma : \{1, \ldots, k\} \to D$ be the store. We define (q, a, I, l, q') as a transition in automaton from state q to state q' upon reading (a, d) if $\sigma[i] = d$ for all $i \in I$ and updates $\sigma[l]$ to d, and define (q, q') as an ϵ-transition that switches the state without reading the next letter.

Definition 62 (Runs of Session Automata)
A *run* of a k-register session automaton $A = (Q, q_0, \Delta, F)$ on a data word

$$w = (a_1, d_1), \ldots, (a_k, d_k) \in (\Sigma \times D)^k$$

is a sequence $s \in \big(Q \times \mathbb{N} \times (\{1, \ldots, k\} \to D)\big)^*$. An element (q, j, σ) of the sequence denotes that A is at state q with store σ and reads (a_j, d_j). To be a run of A, the sequence $s = (q_0, j_0, \sigma_0), \ldots, (q_n, j_n, \sigma_n)$ must satisfy the following for all $i \leq n$:

$$\Big((q_i, q_{i+1}) \in \Delta \wedge (j_i = j_{i+1}) \wedge (\sigma_i = \sigma_{i+1})\Big) \vee$$
$$\Big((q_i, (a_{j_i}, d_{j_i}), I, l, q_{i+1}) \in \Delta \wedge (j_{i+1} = j_i + 1) \wedge (\sigma_{i+1} = \sigma_i[l \mapsto d_{j_i}]) \wedge \forall l \in I.\ \sigma_i(l) = d_{j_i}\Big)$$

Now we fix Σ to be $\Sigma = \big((\{\texttt{start}\} \times \mathsf{Met}) \cup \{\texttt{react}\}\big)$ and $D = \mathsf{Fut}$, where `start` labels process activation and `react` labels process reactivation.

The connection to the rules in Section 2.1 on page 16 is that in a system $(\overrightarrow{Sys}, \overrightarrow{Sch}, h)$ the scheduler is a pair $(A, (q, \sigma))$ with $A = (Q, q_0, \Delta, F)$ and

$$\overrightarrow{Sch}(O)(a, d) = (A, (q', \sigma')) \iff (q, (a, d), I, l, q') \in \Delta \wedge (\sigma' = \sigma[l \mapsto d]) \wedge \forall i \in I.\ \sigma(i) = d$$

The scheduler advances the automaton on every scheduling decision instead of inputting the whole word at once.

Given an object type, we build a session automata from a given object type:

Definition 63
Let $\mathbf{L}$ be an object type. Let k be number of futures in $\mathbf{L}$. The k-register session automaton $\mathfrak{A}(\mathbf{L})$ is defined inductively as follows:

- The receiving action $\mathbf{L} = \mathbf{p}?_f m$ is mapped to a 2-state automaton which reads $(\texttt{start}, m)$ and stores the future f in the $pos(f)$-th register on its sole transition:

$$\mathfrak{A}(\mathbf{L}) = \big(\{q_1, q_2\}, q_1, \{(q_1, (\texttt{start}, m), \emptyset, \mathit{fpos}_{\mathbf{L}}(f), q_2), \{q_2\}\big)$$

- The reactivation action $\mathbf{L} = \mathsf{React}(f)$ is mapped to a 2-state automaton which reads `react` and tests for equality with the $\mathit{fpos}(f)$-th register on its sole transition.

$$\mathfrak{A}(\mathbf{L})\big(\{q_1, q_2\}, q_1, \{(q_1, (\texttt{react}), \{\mathit{fpos}(f)\}, \mathit{fpos}_{\mathbf{L}}(f), q_2), \{q_2\}\big)$$

- Every other action is mapped to the 1-state automaton which accepts only ϵ

$$\mathfrak{A}(\mathbf{L}) = \big(\{q_1\}, q_1, \emptyset, \{q_1\}\big)$$

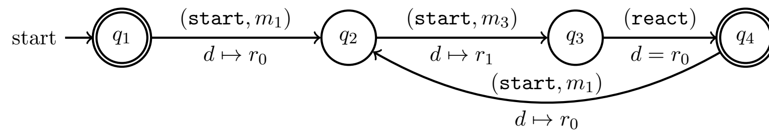
- Concatenation, branching, and repetition using the standard construction for concatenation, union, and repetition for NFAs. We assume that the *fpos* function used in the construction of the subautomata is the one of the whole type.

When a process is activated, the automaton will store the process's corresponding future; and when a process is reactivated, the automaton compares the process's corresponding future with the specified register. As all repetitions in types projected from a global type are self-contained, after the repetition is done, the futures used in the repetition are resolved and thus the automaton can overwrite it safely. The following example shows how a session automaton works based on an object type:

Example 27
Consider the following type:

$$\big(\mathbf{p}?_f m_1\ .\ \mathsf{Await}(f, f')\ .\ \mathbf{p}?_{f''} m_3\ .\ \mathsf{Put}\ f''\ .\ \mathsf{React}(f)\ .\ \mathsf{Put}\ f\big)^*\ .\ \mathsf{end}$$

The generated 2-register session automaton is:



Theorem 3 (Fidelity)
Let S_0 be an ABS system and $\mathbf{G}$ an global type, such that S_0 initially *fits* to $\mathbf{G}$. If for every method m of S_0 there is a proof that $\Phi_{\widetilde{G\upharpoonright_m}}$ is an invariant *and the scheduler of each object* $\mathbf{p}$ *accepts the same language as* $\mathfrak{A}(\mathbf{G}\upharpoonright_{\mathbf{p}})$, then S_0 is captured by $\mathbf{G}$.

It is decidable [5] if two session automata accept the same language.

5. Conclusion

Related Work

Multi-party session types for asynchronous communication have been introduced in [18], exception handling has been studied e.g by [9, 8, 10]. while session types for object-oriented programs has been proposed by several authors with several aims.

The Mool language [17, 7] also projects to method insteads of objects. However, it uses non-uniform objects, i.e. the order of method-calls is programmed as part of the class. This is an encoding of typestate and keeps track of the current state of the global type from the point of view of the object. Our session types are not depending on typestate or scheduling: the ordering of method-calls is handled by projecting to the object first and only as a second step to methods. The typestate approach makes it possible that a verified class has the specified order of method-calls independent of the global type. The verification of the method-local types only guarantees this if projected from an admissible global type. The Moose language [14] does not rely on typestate and also projects on methods, but is channel-based.

Both languages have in common that their session types are channel based and verified with a type system and not with a theorem prover. We do not compare the precision of these approaches, but suppose that the symbolic execution of KeY is able to be more precise, at the possible cost of losing full automatization.

For several other mainstream language like Java [20, 19], Haskell [26] or C [24] a type system was developed which relies on the usage of a specific library for channels. Methods are type checked with respect to correct usage of this library, i.e. the correct calls to send and receive methods. This can not deal with the usage of other means of communication, while session types designed for a language capture all communication.

Future Work

Towards Method Contracts for ABS Method-local session types are verified as class-invariants, despite only reasoning about a single method. The generated formulas $\varphi_{\mathbf{L}}$ can be seen as the post conditions of method-contracts. The additional branches to capture non-specified values in futures are similar to pre-conditions.

The ψ-calculus adds 'ensures' and 'requires' clauses to the π-calculus. This would allow us to remove the current restrictions concerning the heap, as changes of the heap could be expressed as ensures and requires clauses.

Session types for such an extension and the treatment of session types as contracts have been investigated by e.g. [3, 25, 23].

We assume that similar work can be done for Session Types for ABS, using some established elements from the JavaDL version of the KeY prover [2]. The global type would be encoded as a pre- and post-conditions for the method *and its release points*, enhancing the precision of the prover.

Example 28
Consider the global type of Example 20 on page 49.

$$\mathsf{U} = \mathbf{0} \xrightarrow{f_0} \mathbf{A}:m() \ . \ \mathbf{A} \xrightarrow{f} \mathbf{B}:m2() \ . \ \mathsf{Rel}(\mathbf{A}, f) \ . \ \mathbf{B} \xrightarrow{f'} \mathbf{A}:m1() \ . \ \mathbf{A}{\downarrow}f' \ . \ \mathbf{B}{\uparrow}f' \ . \ \mathbf{B}{\downarrow}f \ . \ \mathbf{A}{\downarrow}f_0$$

Consider the case that the method $A.m1$ does not alter the state. When executing according to $\mathbf{G}$, the Method $A.m$ can assume after its reactivation that the heap did not change, because only a pure method is executed in between. Thus when symbolically executing $A.m$ it is not necessary to lose information by anonymizing over the heap.

Towards Composition Our approach is only able to describe the communication in a system from a global view with a fixed, finite number of endpoints.

Many systems, especially those motivated by web services do not describe a global scenario but the local behavior of one endpoint, e.g. BitTorrent [1]. Similarly ABS implementations were done with respect to those local behavior descriptions [27]. Global behavior is ensured by checking whether the local types are compatible, e.g. in the binary setting a check for duality is carried out [29]. A notion of compatibility for multi-party type based on communicating automata has been established in [12, 13].

One line of future work would be to establish a concept of comparability for object-local types for ABS based on [12, 13]. In this work we used automata over infinite alphabets to describe scheduling inside objects. We suppose that communicating automata for infinite alphabets [4, 5] can also model single ABS processes. To adapt the approach of [12], one needs to develop the correct notion of communication between such automata.

Recent case studies [16] for ABS have already carried out compositional reasoning for ABS invariants for arbitrary many endpoints. We hope that a composition of object-local types could be a step towards automatization of such analysis.

A. Appendix

A.1. Proofs

For Theorem 1

If a global type $\mathbf{G}$ is well-formed, then each history in $\mathcal{L}(\tau(\mathbf{G}))$ is well-formed.

Proof. We prove that if there is an ill-formed $h \in \tau(\mathbf{G})$, then $\mathbf{G}$ is not well-formed. As every history is captured by a single branch of $\mathbf{G}$, w.l.o.g. we assume that $\mathbf{G}$ is linear.

We have the following cases for h:

1. There is a position i such that

$$h[i] = \mathsf{invEv}(\mathbf{O}, \mathbf{O}', f, m, e)$$

and there is a position $j < i$ with

$$h[j] = \mathsf{invEv}(\mathbf{O}'', \mathbf{O}''', f, m', e')$$

I.e. the future is not fresh. In this case $\mathbf{G}$ must have the form

$$\mathbf{G}_1 \ . \ \mathbf{O} \xrightarrow{f} \mathbf{O}' \ . \ \mathbf{G}_2 \ . \ \mathbf{O}'' \xrightarrow{f} \mathbf{O}''' \ . \ \mathbf{G}_3$$

In this case $\mathbf{G}$ cannot be well-formed as $\mathbf{G} \upharpoonright_{\mathbf{O}''}$ is undefined because *fresh*(f, k) will not hold at the node k of the AST corresponding to $\mathbf{O}'' \xrightarrow{f} \mathbf{O}'''$.

2. There is a position i such that

$$h[i] = \mathsf{invREv}(\mathbf{O}, \mathbf{O}', f, m, e)$$

but there is no position $j < i$ with

$$h[j] = \mathsf{invEv}(\mathbf{O}, \mathbf{O}', f, m, e)$$

I.e. the process starts without being invoked. This can not be the case, as if $\mathsf{invREv}(\mathbf{O}, \mathbf{O}', f, m, e)$ is in the history, then $\mathbf{G}$ must have the form $\mathbf{G}_1 \ . \ \mathbf{O} \xrightarrow{f} \mathbf{O}' \ . \ \mathbf{G}_2$ and

$$\tau(\mathbf{O} \xrightarrow{f} \mathbf{O}') = [\mathsf{invEv}(\mathbf{O}, \mathbf{O}', f, m, e), \mathsf{invREv}(\mathbf{O}, \mathbf{O}', f, m, e')]$$

. Thus $h \notin \tau(\mathbf{G})$.

3. There is a position i such that

$$h[i] = \mathsf{invREv}(\mathbf{O}, \mathbf{O}', f, m, e)$$

and a position $j < i$ with

$$h[j] = \mathsf{invEv}(\mathbf{O}, \mathbf{O'}, f, m, e)$$

but there is a position k with $i < k < j$ such that $h[k] = \mathsf{invREv}(\mathbf{O}, \mathbf{O'}, f, m, e)$. I.e. the process starts twice. This is not possible, because if there are two positions with the event $\mathsf{invREv}(\mathbf{O}, \mathbf{O'}, f, m, e)$, then $\mathbf{G}$ must have the form

$$\mathbf{G}_1 \;.\; \mathbf{O} \xrightarrow{f} \mathbf{O'} \;.\; \mathbf{G}_2 \;.\; \mathbf{O} \xrightarrow{f} \mathbf{O'} \;.\; \mathbf{G}_3$$

Thus the future is not fresh and $\mathbf{G}$ is not well-formed as in case 1.

4. There is a position i such that

$$h[i] = \mathsf{futEv}(\mathbf{O}, f, m, e)$$

but there is no position $j < i$ with

$$h[j] = \mathsf{invREv}(\mathbf{O'}, \mathbf{O}, f, m, e')$$

I.e. the process terminates without being started. In this case $\mathbf{G}$ must have the form $\mathbf{G}_1 \;.\; \mathbf{O}{\downarrow}f \;.\; \mathbf{G}_2$ and $\mathbf{G}_1$ does not contain $\mathbf{O} \xrightarrow{f} \mathbf{O'}$. Then $\mathbf{G}$ is not well-formed because $\mathbf{G} \upharpoonright_{\mathbf{O}} \upharpoonright_f$ is undefined at $\mathbf{O}{\downarrow}f$, because if $\mathbf{G}_1$ does not contain $\mathbf{O} \xrightarrow{f} \mathbf{O'}$, f can never be active.

5. There is a position i such that

$$h[i] = \mathsf{futEv}(\mathbf{O}, f, m, e)$$

and a position $j < i$ with

$$h[j] = \mathsf{futEv}(\mathbf{O}, f, m, e') \vee h[j] = \mathsf{throwEv}(\mathbf{O}, f, m, e')$$

I.e. the process is terminated twice. In this case $\mathbf{G}$ must have the form

$$\mathbf{G}_1 \;.\; \mathbf{O}{\downarrow}f \;.\; \mathbf{G}_2 \;.\; \mathbf{O}{\downarrow}f \;.\; \mathbf{G}_3$$

. The $\mathbf{G}$ is not well-formed, because after $\mathbf{O}{\downarrow}f \;.\; \mathbf{G}_2$ the future f can not be active anymore and thus $\mathbf{G} \upharpoonright_{\mathbf{O}} \upharpoonright_f$ is undefined at the second $\mathbf{O}{\downarrow}f$.

6. The last two cases are analogous for $h[i] = \mathsf{throwEv}(\mathbf{O}, f, m, e)$ and $h[i] = \mathsf{awaitEv}(\mathbf{O}, f, f')$.

7. There is a position i such that

$$h[i] = \mathsf{futREv}(\mathbf{O}, f, e)$$

but there is no position $j < i$ with

$$h[j] = \mathsf{futEv}(\mathbf{O}, f, m, e')$$

In this case $\mathbf{G}$ must have the form $\mathbf{G}_1 \;.\; \mathbf{O}{\uparrow}f \;.\; \mathbf{G}_2$ and $\mathbf{G}_1$ does not contain $\mathbf{O}{\downarrow}f$. Then $\mathbf{G}$ is not well-formed, as this is explicitly checked when computing $\mathbf{G} \upharpoonright_{\mathbf{O}}$

8. The above case it analogous for $h[i] = \mathsf{throwREv}(\mathbf{O}, f, e)$.

This concludes the proof. □

For Lemma 1

A well-formed global type $\mathbf{G}$ is not admissible, iff some type in $u_{0,2}(\mathbf{G})$ is not admissible.

Proof.
- The "only if" direction follows directly from

$$\bigcup_{\mathbf{G}' \in u_{0,2}(\mathbf{G})} \mathcal{L}(\tau(\mathbf{G}')) \subseteq \mathcal{L}(\tau(\mathbf{G}))$$

If there is a well-formed h in some unrolling of $\mathbf{G}$ in $u_{0,2}(\mathbf{G})$, that has a non-local-order preserving permutation, then this h is also in $\mathcal{L}(\tau(\mathbf{G}))$.

- For the other direction, consider the following induction on $sh(\mathbf{G})$
  - **Induction Base** $sh(\mathbf{G}) = 0$:

    In this case $u_{0,2}(\mathbf{G}) = \{\mathbf{G}\}$.
  - **Induction Step** $sh(\mathbf{G}) = k + 1$

    In this case $\mathbf{G}$ has the form $\mathbf{G}_1 \ . \ (\mathbf{G}_2)^* \ . \ \mathbf{G}_3$ for some $\mathbf{G}_1, \mathbf{G}_2, \mathbf{G}_3$ with $sh(\mathbf{G}_1) = sh(\mathbf{G}_2) = sh(\mathbf{G}_3) = k$

    Let $\mathbf{G}' \in u_{0,2}(\mathbf{G})$ be a type, such that there is a $h \in \mathcal{L}(\tau(\mathbf{G}'))$ that has a non-local-order preserving permutation. The history h must have the form $h = h_1 \circ h_2 \circ h_3$ with $h_1 \in \mathcal{L}(\tau(\mathbf{G}_1))$, $h_2 \in \mathcal{L}(\tau(\mathbf{G}_2^*))$ and $h_3 \in \mathcal{L}(\tau(\mathbf{G}_3))$.

    Any axiom of well-formedness does not hold for a history h, only if there are $i, j < |h|$ with $i < j$ such that $h[i]$ breaks some conditions imposed by $h[j]$. Thus a permutation h' of h can only be non-local-order preserving if it mapped $h[i]$ to some $h'[i']$ and $h[j]$ to some $h'[j']$ with $j' < i'$.

    We have to show that if there is such a h and a permutation h' in $\mathcal{L}(\tau(\mathbf{G}))$, then there is one for some element of $u_{0,2}(\mathbf{G})$. We make a case distinction on the position of i and j.

    * **Case i and j are both in h_1**

      In this case, we can construct a non-local-order preserving permutation h_1' of h_1 such that $h_1' \circ h_2 \circ h_3$ is a non-local-order preserving permutation of h by induction hypothesis. Now as $\mathbf{G}_2$ is self-contained, $h_1 \circ h_3$ is also a well-formed history in $\mathcal{L}(\tau(\mathbf{G}))$ and thus $h_1' \circ h_3$ is a non-local-order preserving permutation of it. The history $h_1 \circ h_3$ is described by$\mathbf{G}_1 \ . \ (\mathbf{G}_2)^0 \ . \ \mathbf{G}_3$ and this type is in $u_{0,2}(\mathbf{G})$.

$$h_1 \circ h_3 \in \mathcal{L}(\tau(\mathbf{G}_1 \ . \ (\mathbf{G}_2)^0 \ . \ \mathbf{G}_3))$$
$$\mathbf{G}_1 \ . \ (\mathbf{G}_2)^0 \ . \ \mathbf{G}_3 \in u_{0,2}(\mathbf{G})$$

    * **Case i and j are both in h_3**

      Analogous to the above case.
    * **Case i and j are both in h_2**

      $h_2 = g_1 \circ \cdots \circ g_l$ is a l-fold repetition for some $g_1, \ldots, g_l$ which are all pair-wise future-equivalent. Here we have to distinguish two cases

· i and j are in the same g_n subsequence. This case is analogous to the two above cases.

· i and j are in different subsequences $g_n, g_m, n < m < l$ As the subsequences are all pair-wise future-equivalent, if we can find such a pair i, j for some n, m, then we can such pair for any n, m, especially for $n = 1$ and $m = 2$. I.e. $h_2 = g_1 \circ g_2$. Then

$$h \in \mathcal{L}(\tau(\mathbf{G}_1 \ . \ (\mathbf{G}_2)^2 \ . \ \mathbf{G}_3)) \in u_{0,2}(\mathbf{G})$$

∗ **Case i is in h_1 and j is in h_2**

In this case $h_2 = g_1 \circ \cdots \circ g_l$ is a l-fold repetition for some $g_1, \ldots, g_l$ which are all pair-wise future-equivalent. Let j be in some g_n. I.e. the change of order of $h[i]$ and $h[j]$ breaks well-formedness for some axiom. As all subsequences are pair-wise future-equivalent and result from a a self-contained type, we can find such a $h[j]$ in every $g_n, n < l$, especially for g_1 I.e. $h_2 = g_1 \circ g_2$. Now with $h = h_1 \circ g_1 \circ g_2 \circ h_3$:

$$h \in \mathcal{L}(\tau(\mathbf{G}_1 \ . \ (\mathbf{G}_2)^2 \ . \ \mathbf{G}_3)) \in u_{0,2}(\mathbf{G})$$
$$\mathbf{G}_1 \ . \ (\mathbf{G}_2)^2 \ . \ \mathbf{G}_3 \in u_{0,2}(\mathbf{G})$$

∗ **Case i is in h_2 and j is in h_3**

Analogous to the above case.

∗ **Case i is in h_1 and j is in h_3**

Es shown in Example 19, there are two sub cases:

∗ $h \in \mathcal{L}(\tau(\mathbf{G}_1 \ . \ (\mathbf{G}_2)^0 \ . \ \mathbf{G}_3))$: I.e. the change of order can only happen if the repetition is not executed. This holds because $\mathbf{G}_1 \ . \ (\mathbf{G}_2)^0 \ . \ \mathbf{G}_3 \in u_{0,2}(\mathbf{G})$

∗ $h \in \mathcal{L}(\tau(\mathbf{G}_1 \ . \ (\mathbf{G}_2)^n \ . \ \mathbf{G}_3))$, for some $n > 0$ If some position in h_3 can be permuted before some position h_1 but the break of well-formedness occurs independent of anything in h_2, then there is some $h'' \in \mathcal{L}(\tau(\mathbf{G}_1 \ . \ (\mathbf{G}_2)^2 \ . \ \mathbf{G}_3))$ which has a non-local-order preserving permutation. □

For Lemma 2

Let $\mathbf{G}$ be a well-formed linear global type which contains no unbounded repetition and $\mathfrak{C}(\mathbf{G})$ its causality graph. Let $I(P)$ be the set of indices where $\tau(\mathbf{G})[i] \upharpoonright_P \neq \epsilon$ for an object P.

If for each P for each $i, j \in I(P), i < j$, where i, j are in the same connected component, there is a path from i to j in $\mathfrak{C}(\mathbf{G}) = (V_\mathbf{G}, E_\mathbf{G})$.

$$\forall P \in \mathsf{Ob}. \ \forall i, j \in I(P). \ i < j \rightarrow E_\mathbf{G}(i, j)$$

Then $\mathbf{G}$ is admissible.

Proof.
Let h be the translation of $\mathbf{G}$, and $\mathbf{O} \in \mathsf{Ob}$ any object. It suffices to show that if there is a path p from i to j for $i, j \in I(\mathbf{O})$, and if $h[i]$ and $h[j]$ are swapped then the resulting history h' is either not well-formed or $h \upharpoonright_\mathbf{O} \neq h' \upharpoonright_\mathbf{O}$.

The proof is by induction on the length of the path between i and j

Induction Base length = 1: Case distinction why this edge is in the graph: We have the following cases why this edge is in the graph, by Definition 52:

- Condition 4.2 can not be violated, because if $i, j \in I(\mathbf{O})$, both events are issued by the same object and self-call do not give rise to events.
- By condition 4.3: $h[i] = \mathsf{futEv}(P, f, m, \epsilon) \wedge h[j] = \mathsf{futREv}(Q, f, \epsilon)$ The swapping of a resolving and the corresponding fetching event does not result in a well-formed history.
- By condition 4.4: $h[i] = \mathsf{throwEv}(P, f, m, \epsilon) \wedge h[j] = \mathsf{throwREv}(Q, f, \epsilon)$ The swapping of a throwing and the corresponding catching event does not result in a well-formed history.
- By condition 4.5: $h[i]$ and $h[j]$ are issued by the same object. In this case swapping them would break the condition $h\restriction_O = h'\restriction_O$.
- By condition 4.8: $h[i] = \mathsf{futEv}(P, f, m, \epsilon) \vee h[i] = \mathsf{throwEv}(P, f, m, \epsilon)$ and $h[j]$ is the first event after a reactivation, triggered by $h[i]$. As $i, j \in I(\mathbf{O})$, both events are issued by the same object and swapping them would break the condition $h\restriction_O = h'\restriction_O$.

Induction Step length > 1: There is a k with $i < k < j$ with two cases

1. There is a path from i to k and an *edge* from k to j.
2. There is an *edge* from i to k and a path from k to j

In both cases, we can not reorder k before i and after j by induction hypothesis. The additional edge forbids the swapping of i and k, resp. k and j. We make a case distinction why the edge is added to the graph to show that we can also not swap i and j.

- Conditions 4.3, 4.4 and 4.5 are as in the induction base. If it is not possible to swap i after k, it is also not possible to swap i for any position bigger than k, especially j.
- By condition 4.2: $h[k] = \mathsf{invEv}(P, Q, f, m, e) \wedge h[j] = \mathsf{invREv}(P, Q, f, m, e)$ (or vice versa). The swapping of a invocation and the corresponding invocation reaction event does not result in a well-formed history. If it is not possible to swap i after k, it is also not possible to swap i for any position bigger than k, especially j.
- By condition 4.8: $h[k] = \mathsf{futEv}(P, f, m, \epsilon) \vee h[k] = \mathsf{throwEv}(P, f, m, \epsilon)$ and $h[j]$ is the first event after a reactivation, triggered by $h[k]$ (or v.v.). Obviously $h[i]$ and $h[j]$ can still not be swapped due to being issued by the same object (but possibly different processes). The difference to previous cases is that it is perfectly fine to swap $h[k]$ and $h[j]$, because they are from different objects.

Now, if for all pairs $i, j \in I(\mathbf{O})$ with $i < j$ there is a path, no swapping in $h\restriction_O$ can occur and thus $\mathbf{G}$ is admissible. □

Auxiliary Lemmas

To prove the fidelity we need two auxiliary lemmas.

First, we need to show that the translation $T(\mathbf{L})$ of a linear repetition-free well-formed method-local type $\mathbf{L}$ into an ABSDL term and an ABSDL formula encodes $\tau(\mathbf{L})$ correctly. Recall that $\mathsf{val}_{C,\beta}$ is the evaluation function of ABSDL that maps formulas to truth values and terms to domain elements. The function β is the assignment of logical variables that maps variables to domain elements of fitting type.

Lemma 3 (Correctness of $T(\mathbf{L}) = (t_{\mathbf{L}}, \psi_{\mathbf{L}})$)
Let $\mathbf{L}$ be a linear repetition-free method-local type. The translation $T(\mathbf{L})$ encodes $\mathbf{L}$ correctly:

$$\mathsf{val}_{C,\beta}(t \doteq t_{\mathbf{L}} \wedge \psi_{\mathbf{L}}) = \mathbf{tt} \rightarrow \beta(t) \in \mathcal{L}(\tau(\mathbf{L}))$$

Proof. The proof is by induction on the structure of $\mathbf{L}$. We abstain from explicitly giving the case distinction, as the interpretation $\mathcal{I}$ maps every function symbol for an event to the corresponding event (and propagates the arguments).

Secondly, we need to show that the translation of a well-formed method-local type $\mathbf{L}$ into an ABSDL formula $\varphi_{\mathbf{L}}(v)$ with free variable v encodes $\tau(\mathbf{L})$ correctly, i.e. if $\varphi_{\mathbf{L}}(v)$ holds then v must be evaluated to a an element of $\mathcal{L}(\tau(\mathbf{L})))$.

Lemma 4 (Correctness of $\varphi_{\mathbf{L}}$)
Let $\mathbf{G}$ be a well-formed global type and $\mathbf{L} = \mathbf{G} \upharpoonright_m$ for some method m. If $\mathsf{val}_{C,\beta}(\varphi_{\mathbf{L}}(v))$ holds, then $\mathsf{val}_{C,\beta}(v) \in \mathcal{L}(\tau(\mathbf{L})))$.

Proof. The formula has the form $\varphi_{\mathbf{L}}(v) = \bigvee_{\mathbf{L}' \in T} \varphi_{\mathbf{L}'}(v)$ for some linear $\mathbf{L}'$, which all have the same star-height. We fix one $\mathbf{L}'$ such that $\mathsf{val}_{C,\beta}(\varphi_{\mathbf{L}'}(v))$ holds.

The only free variable in $\varphi_{\mathbf{L}'}(v)$ is v, so we can assume a β' such that $\mathsf{val}_{C,\beta'}(\chi_{\mathbf{L}'})$ holds with $\mathsf{val}_{C,\beta'}(f_i) \neq \mathsf{val}_{C,\beta'}(f_j)$ for every futures f_i, f_j occurring in $\mathbf{L}'$. Induction over the star-height of $\mathbf{L}$.

- **Induction Base** $sh(\mathbf{L}) = 0$:

  Now as $\mathbf{L}'$ is repetition-free and linear, its disassembly (see Definition 57 is

  $$\begin{aligned} \mathfrak{T}(\mathbf{L}') &= (T, R, B) \\ T &= \{(v, \mathbf{L}')\} \\ R &= \emptyset \\ \mathbf{dom}(B) &= \emptyset \end{aligned}$$

  Thus

  $$\chi_{\mathbf{L}'} = v \doteq t_{\mathbf{L}'} \wedge \psi_{\mathbf{L}'}$$

  and by Lemma 3 follows that if $v \doteq t_{\mathbf{L}'} \wedge \psi_{\mathbf{L}'}$ holds in some C, β', then $\mathsf{val}_{C,\beta'}(v) \in \mathcal{L}(\tau(\mathbf{L})))$.

- **Induction Step** $sh(\mathbf{L}) = k + 1$.

  As $sh(\mathbf{L}) > 0$, we can assume the type $\mathbf{L}$ has the form

  $$\mathbf{L} = \mathbf{L}_1 \ . \ (\mathbf{L}_2)^* \ . \ \mathbf{L}_3$$

  for some $\mathbf{L}_1, \mathbf{L}_2, \mathbf{L}_3$ with $sh(\mathbf{L}_1) = sh(\mathbf{L}_2) = sh(\mathbf{L}_3) \leq k$.

  The proof also holds if more than one type with star-height k is repeated, i.e. if $\mathbf{L}$ has the form $\mathbf{L}_1 \ . \ (\mathbf{L}_2)^* \ . \ \mathbf{L}_3 \ . \ (\mathbf{L}_4)^* \ . \ \mathbf{L}_5 \ldots$ with $sh(\mathbf{L}_i) \leq k$.

  We make a case distinction whether $\mathbf{L}_2$ is linear

- $\mathbf{L}_2$ is linear.

  Every history $h \in \mathcal{L}(\tau(\mathbf{L}))$ must have the form $h_1 \circ h_2 \circ h_3$ Now its disassembly is

$$\begin{aligned}\mathfrak{T}(\mathbf{L}) &= (T, R, B)\\ T &= T'[t_2 \mapsto \mathbf{L}_2']\\ R &= R_1 \cup R_2 \cup R_3 \cup \{t_2\}\end{aligned}$$

  for some B, R_1, R_2, R_3, T' with $B(t_2) = \emptyset$, where $\mathbf{L}_2'$ is the flattening of $\mathbf{L}_2$. The encoding formula has the following form:

$$\chi_{\mathbf{L}} = \chi_{\mathbf{L}_1} \wedge \chi_{\mathbf{L}_3} \wedge \mathit{pattern}(t_2, t_{\mathbf{L}_2'}, i) \wedge \chi_{\mathbf{L}_2'}$$

  Where $\chi_{\mathbf{L}_2'}$ is a conjunction of $\psi_{\mathbf{L}_2'}$ and formulas describing the repetition removed from $\mathbf{L}_2$.

  By induction hypothesis, if $\chi_{\mathbf{L}_1}$ and $\chi_{\mathbf{L}_2'}$ hold, then the prefixes are encoded correctly

$$\begin{aligned}\mathsf{val}_{C,\beta}(v) &= h_1 \circ h_2 \circ h_3\\ h_1 &\in \mathcal{L}(\tau(\mathbf{L}_1))\\ h_3 &\in \mathcal{L}(\tau(\mathbf{L}_3))\end{aligned}$$

  It remains to show that $\mathit{pattern}(t_2, t_{\mathbf{L}_2'}, i) \wedge \psi_{\mathbf{L}_2'}$ encodes $h_2 = \mathsf{val}_{C,\beta}(t_2') \in \mathcal{L}(\tau(\mathbf{L}_2^*))$. By definition of *pattern* if $\mathsf{val}_{C,\beta}(\mathit{pattern}(t_2, t_{\mathbf{L}_2'}, i)) = \mathbf{tt}$ holds, then

$$\mathsf{val}_{C,\beta}(t_2) \equiv_{\mathsf{Fut}} \mathsf{val}_{C,\beta}(t_{\mathbf{L}_2'})^i$$

  As additionally $\psi_{\mathbf{L}_2'} \wedge \chi_{\mathbf{L}_2}$ holds, by Lemma 3 $t_{\mathbf{L}_2}$ and induction hypothesis:

$$\begin{aligned}\mathsf{val}_{C,\beta}(t_2) &\equiv_{\mathsf{Fut}} h'^i\\ h' &\in \mathcal{L}(\tau(\mathbf{L}_2))\end{aligned}$$

  Thus

$$h_2 \in \mathcal{L}(\tau(\mathbf{L}_2^i)) \subseteq \mathcal{L}(\tau(\mathbf{L}_2^*))$$

- $\mathbf{L}_2$ is not linear. The only difference to the previous case is that in the disassembly, $B(t_2)$ is not empty. Then $\chi_{\mathbf{L}}$ has additional conjuncts encoding the branches of $\mathbf{L}_2$. As all the branches also have star-height of k or lower, by induction hypothesis we can construct χ formulas for them and the rest of the proof is analogous. □

For Theorem 2

Let S_0 be an ABS system and $\mathbf{G}$ admissible global type, such that S_0 initially *fits* to $\mathbf{G}$. If for every method m of S_0 there is a proof that $\Phi_{\widetilde{G\upharpoonright_m}}$ is an invariant, then S_0 is captured by $\mathbf{G}$.

Proof. The main step in this proof is showing that the following statement holds:

Let $(S_0, \dots, S_n)$ be a sequence of systems, such that $S_i \to S_{i+1}, i < n$ and S_n is terminated. At every $i < n$, the history of S_i is a prefix of some permutation of some element of $\mathcal{L}(\tau(\mathbf{G}))$.

$$\forall i < n.\ \exists h, h' \in \mathsf{Ev}^*.\ \mathit{his}(S_i) \sqsubseteq h \land \pi(h, h') \land h' \in \mathcal{L}(\tau(\mathbf{G})) \qquad (\star)$$

This means that there is a history h in $\mathcal{L}(\tau(\mathbf{G}))$ that the system builds step by step. Given the above statement, it remains to show that if a system terminates it has not realized some prefix h' of some history h in $\mathcal{L}(\tau(\mathbf{G}))$, but indeed the whole history h.

Induction on the length of $(S_0, \dots, S_n)$ to prove $\star$

- **Induction Base** $n = 0$: As $\mathbf{G}$ is well-formed it has the form

$$\mathbf{0} \xrightarrow{f} \mathbf{A}{:}m\ .\ \mathbf{G}'$$

For some $\mathbf{A}, f, m$. As S_0 initially fits to $\mathbf{G}$, its history is a singleton

$$[\mathsf{invREv}(\mathbf{0}, \mathbf{A}, m, f, \vec{e})]$$

This must be the prefix of any history in $\mathcal{L}(\tau(\mathbf{G}))$

- **Induction Step** $n > 0$ We have to show that if at position $i < n$, the history of S_i is a prefix of some permutation of some element of $\mathcal{L}(\tau(\mathbf{G}))$, then the history of S_{i+1} is also a prefix of some permutation of some element of $\mathcal{L}(\tau(\mathbf{G}))$.

$$\forall i < n.\ \Big(\big(\exists h, h' \in \mathsf{Ev}^*.\ \mathit{his}(S_i) \sqsubseteq h \land \pi(h, h') \land h' \in \mathcal{L}(\tau(\mathbf{G}))\big)$$
$$\to \big(\exists h, h' \in \mathsf{Ev}^*.\ \mathit{his}(S_{i+1}) \sqsubseteq h \land \pi(h, h') \land h' \in \mathcal{L}(\tau(\mathbf{G}))\big)\Big)$$

First, it is not possible that $\mathit{his}(S_i)\!\restriction_O$ is the prefix of some permutation of some history in $\mathcal{L}(\tau(\widetilde{\mathbf{G}\!\restriction_O})) \setminus \mathcal{L}(\tau(\mathbf{G}\!\restriction_O))$: If this would be the case, there must be a position k with

$$\mathit{his}(S_i)[k] = \mathsf{futREv}(\mathbf{O}, f, C(\vec{e}))$$

or

$$\mathit{his}(S_i)[k] = \mathsf{throwREv}(\mathbf{O}, f, C(\vec{e}))$$

such that the outermost constructor C is a label for one of the new added branches in $\tilde{\mathbf{L}}$. Then, by well-formedness there must be a $l < k$, such that

$$\mathit{his}(S_i)[l] = \mathsf{futEv}(\mathbf{O}', f, m, C(\vec{e}))$$

or

$$\mathit{his}(S_i)[l] = \mathsf{throwEv}(\mathbf{O}', f, m, C(\vec{e}))$$

I.e. another process terminated and the return value/thrown exception has the required label. However if this results from an *added* branch, there is no action in the type of any other method that translates to such an event.

We make a case distinction over the rule enabling $S_i \to S_{i+1}$

- **internal**: As $\mathit{his}(S_i) = \mathit{his}(S_{i+1})$, the property trivially holds. Note that the awaitEv event encodes control release according to the *act* functions, which we used to determine whether an object is active.
- **inner**: In this case $\mathit{his}(S_{i+1}) = \mathit{his}(S_i) \circ [ev]$ and ev is not an invocation reaction event. Let O be the object and p the process of O issuing ev. Let p be computing the method m and $\mathbf{L} = \mathbf{G} \upharpoonright_m$ it local type. As $\Phi_{\tilde{\mathbf{L}}}$ is an invariant, by Lemma 4 and the above observation, every history m realizes is in $\mathcal{L}(\tau(\mathbf{L}))$.

  Obviously, $\mathit{his}(S_i) \upharpoonright_{O'} = \mathit{his}(S_{i+1}) \upharpoonright_{O'}$ for any object but O. The process p was already active in S_i. Let h, h' be two histories as required by $\star$, such that $\mathit{his}(S_i) \sqsubseteq h$. We distinguish two cases

  * ev **does not correspond to the first action after an offering operator:**

    As $\Phi_{\tilde{\mathbf{L}}}$ is an invariant, by Lemma 4 and the above observation, every step that the process p can make, is captured by some branch in the type. Let h, h' be two histories with $\mathit{his}(S_i) \sqsubseteq h, \pi(h, h')$ and $h' \in \mathcal{L}(\tau(\mathbf{L}))$. We can choose h' according to the branch of $\mathbf{L}$, that has been executed so far and that has ev as its next action. Such a branch must exist, otherwise the $\Phi_{\tilde{\mathbf{L}}}$ would not be an invariant. For this history h', we can again use h as the permutation and $\mathit{his}(S_{i+1}) \sqsubseteq h$ holds.
  * ev **corresponds to the first action after an offering operator:**

    Thus it corresponds to a Get $f : C$ action in the type and p evaluates a f.**get** expression. By Lemma 3 and well-formedness of $\mathbf{G}$, this is possible because p can access f. There is a proof tree that $\Phi_{\tilde{\mathbf{L}}}$ is an invariant. KeY-ABS splits the proof when symbolically executing the read access to futures. The condition that the outermost constructor has a certain form has been proven for all branches, thus the method has an execution path for every offering branch in $\mathbf{L}$, especially for the one which contains the corresponding action.

    If it is ensured that there is some $h' \in \mathcal{L}(\tau(\mathbf{L}))$ that realizes the correct branch, the rest of this case is analogous to the other case.
- **start** This rule starts a new process in some object O. The added event ev is an invocation reaction event. The invocation event that is needed to invoke this rule is some element of h, and h is a prefix of a history accepted by $\mathbf{G}$, thus ev must be added at some point to the history and is captured by the type. It remains to show that the correct process is started, i.e. $(h \circ [ev]) \upharpoonright_O$ is a prefix of the same history as h.

  Suppose this would not be the case. The object O must be inactive, so the possible actions are reactivation or process starting. If there is even a choice, $\mathbf{G}$ would not be well-formed, thus if this rule has been applied, it starts the correct process.
- **continue** This case is analogous to the case for **startActive**: as $\mathbf{G}$ is admissible, at every point an object can activate or start a process it is enforced by callee and suspension guards that there is no choice possible. Also reactivation does not add an event to the history: $\mathit{his}(S_i) = \mathit{his}(S_{i+1})$.

From this induction follows that $\mathit{his}(S_n)$ is the prefix of some local-order preserving permu-

tation of some history described by $\mathbf{G}$:

$$\exists h, h' \in \mathsf{Ev}^*.\ \mathit{his}(S_n) \sqsubseteq h \wedge \pi(h, h') \wedge h' \in \mathcal{L}(\tau(\mathbf{G}))$$

It remains to show that S_n describes the whole history, i.e. $h \sqsubseteq \mathit{his}(S_n)$ holds and thus $h = \mathit{his}(S_n)$.

Suppose this would not be the case. Then there would be some non-empty suffix g such that $h = \mathit{his}(S_n) \circ g$. The events in g must be issued by some process p in some object O. Especially the resolving/throwing event of p must yet be added to the history. As S_n is terminated, all processes are terminated, no process is suspended and there is no further step possible, i.e. there is no invocation in the history without corresponding invocation reaction event. In this situation, $\mathbf{G}$ has a node i in its AST, such that $\mathit{act}(i, o) = \bot$ but $\mathsf{action}(i) \neq \mathsf{end}$. But at this point no projection to a object-local type is defined and $\mathbf{G}$ would not be well-formed. This is a contradiction. □

For Theorem 3

> Let S_0 be an ABS system and $\mathbf{G}$ an global type, such that S_0 initially *fits* to $\mathbf{G}$. If for every method m of S_0 there is a proof that $\Phi_{\widetilde{G\restriction_m}}$ is an invariant *and the scheduler of each object* $\mathbf{p}$ *accepts the same language as* $\mathfrak{A}(\mathbf{G}\restriction_{\mathbf{p}})$, then S_0 is captured by $\mathbf{G}$.

Proof. The proof is analogous to the one for Theorem 2, except for the justification why the rule **start** and **continue** preserve $(*)$ (i.e. **start** adds the correct event to the history and **continue** reactivated the correct process):

Suppose **start** would add the wrong event $\mathsf{invREv}(O, O', f, m, \vec{e})$, i.e. start the wrong process. The object was inactive before, thus there is a correct process that can be activated (not necessarely now) with an event $\mathsf{invREv}(O, O', f', m', \vec{e}')$ or reactivated (without event). By premisse the scheduler accepted the activation of this process, i.e. the input $(\mathtt{start}, m, f)$. But the scheduler encodes the order of invocation reaction events: If two invREv events $\mathsf{invREv}(O, O', f, m, \vec{e}), \mathsf{invREv}(O, O', f', m', \vec{e}')$ are in some order in a history $h \in \mathcal{L}(\tau(\mathbf{G}))$, then they are letters $(\mathtt{start}, m, f), (\mathtt{start}, m', f')$ in the same order on m, m' in some $w \in \mathcal{L}(\mathfrak{A}(\mathbf{G}\restriction_{\mathbf{O}}))$. Thus if the scheduler accepts the query $(\mathtt{start}, m, f)$ can not be wrong. The case for **continue** is analogous. □

Bibliography

[1] The BitTorrent protocol specification. http://www.bittorrent.org/beps/bep_0003.html. Accessed: 05.02.2016.

[2] Beckert, B., Hähnle, R., and Schmitt, P. H. *Verification of Object-oriented Software: The KeY Approach*, vol. 4334 of *LNCS*. Berlin, Heidelberg, 2007.

[3] Bocchi, L., Honda, K., Tuosto, E., and Yoshida, N. A theory of design-by-contract for distributed multiparty interactions. In *CONCUR'10* (2010), vol. 6269 of *LNCS*, Springer, pp. 162–176.

[4] Bojańczyk, M., David, C., Muscholl, A., Schwentick, T., and Segoufin, L. Two-variable logic on data words. *ACM Trans. Comput. Logic 12*, 4 (July 2011), 27:1–27:26.

[5] Bollig, B., Habermehl, P., Leucker, M., and Monmege, B. A fresh approach to learning register automata. In *DLT'13* (2013), M. Béal and O. Carton, Eds., vol. 7907 of *LNCS*, Springer, pp. 118–130.

[6] Büchi, J. R. Symposium on decision problems: On a decision method in restricted second order arithmetic. In *Logic, Methodology and Philosophy of Science Proceeding of the 1960 International Congress* (1966), P. S. Ernest Nagel and A. Tarski, Eds., vol. 44 of *Studies in Logic and the Foundations of Mathematics*, Elsevier, pp. 1 – 11.

[7] Campos, J., and Vasconcelos, V. T. Channels as objects in concurrent object-oriented programming. In *PLACES'10* (2010), K. Honda and A. Mycroft, Eds., vol. 69 of *EPTCS*, pp. 12–28.

[8] Carbone, M., Honda, K., and Yoshida, N. Structured interactional exceptions in session types. *LNCS 5201 LNCS* (2008), 402–417.

[9] Chen, T., Viering, M., Bejleri, A., Ziarek, L., and Eugster, P. A type theory for robust failure handling in distributed systems. In *FORTE'16, Proceedings* (2016), E. Albert and I. Lanese, Eds., vol. 9688 of *Lecture Notes in Computer Science*, Springer, pp. 96–113.

[10] Collet, R., and Roy, P. *Advanced Topics in Exception Handling Techniques.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, ch. Failure Handling in a Network-Transparent Distributed Programming Language, pp. 121–140.

[11] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[12] Deniélou, P.-M., and Yoshida, N. Multiparty session types meet communicating automata. In *Proceedings of the 21st European Conference on Programming Languages and Systems* (Berlin, Heidelberg, 2012), ESOP'12, Springer-Verlag, pp. 194–213.

[13] Deniélou, P.-M., and Yoshida, N. Multiparty Compatibility in Communicating Automata: Characterisation and Synthesis of Global Session Types. In *Proceedings of the 40th International Conference on Automata, Languages, and Programming - Volume Part II* (Berlin, Heidelberg, 2013), ICALP'13, Springer-Verlag, pp. 174–186.

[14] Dezani-Ciancaglini, M., Drossopoulou, S., Mostrous, D., and Yoshida, N. Objects and session types. In *Inf. Comput.* (May 2009), vol. 207, Academic Press, Inc, pp. 595–641.

[15] Din, C. C., Bubel, R., and Hähnle, R. KeY-ABS: A Deductive Verification Tool for the Concurrent Modelling Language ABS. In *Automated Deduction - CADE'25*, A. P. Felty and A. Middeldorp, Eds., vol. 9195 of *Lecture Notes in Computer Science*. 2015, pp. 517–526.

[16] Din, C. C., Tapia Tarifa, S. L., Hähnle, R., and Johnsen, E. B. Formal Methods and Software Engineering: 17th International Conference on Formal Engineering Methods, ICFEM'15, Proceedings. M. Butler, S. Conchon, and F. Zaïdi, Eds., Springer International Publishing, pp. 217–233.

[17] Gay, S. J., Gesbert, N., Ravara, A., and Vasconcelos, V. T. Modular session types for objects. *Logical Methods in Computer Science 11*, 4 (2015).

[18] Honda, K., Yoshida, N., and Carbone, M. Multiparty asynchronous session types. *SIGPLAN Not. 43*, 1 (Jan. 2008), 273–284.

[19] Hu, R., Kouzapas, D., Pernet, O., Yoshida, N., and Honda, K. Type-safe eventful sessions in java. In *ECOOP'10 Object-Oriented Programming*, T. D'Hondt, Ed., vol. 6183 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2010, pp. 329–353.

[20] Hu, R., Yoshida, N., and Honda, K. Session-based distributed programming in java. In *Proceedings of the 22nd European Conference on Object-Oriented Programming* (Berlin, Heidelberg, 2008), ECOOP'08, Springer-Verlag, pp. 516–541.

[21] Johnsen, E. B., Hähnle, R., Schäfer, J., Schlatte, R., and Steffen, M. Formal Methods for Components and Objects: 9th International Symposium, FMCO 2010, Graz, Austria, November 29 - December 1, 2010. Revised Papers. B. K. Aichernig, F. S. Boer, and M. M. Bonsangue, Eds., Springer Berlin Heidelberg, pp. 142–164.

[22] Kaminski, M., and Francez, N. Finite-memory automata. *Theor. Comput. Sci. 134*, 2 (1994), 329–363.

[23] Laneve, C., and Padovani, L. The Pairing of Contracts and Session Types. In *Concurrency, Graphs and Models (Ugo65'08)* (2008), vol. 5065 of *LNCS*, Springer, pp. 681–700.

[24] Ng, N., Yoshida, N., and Honda, K. Multiparty session c: Safe parallel programming with message optimisation. In *TOOLS 2012* (2012), vol. 7304 of *LNCS*, Springer, pp. 202–218.

[25] Padovani, L. Contract-based discovery of web services modulo simple orchestrators. *Theor. Comput. Sci. 411*, 37 (Aug. 2010), 3328–3347.

[26] Pucella, R., and Tov, J. A. Haskell session types with (almost) no class. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell* (New York, NY, USA, 2008), Haskell '08, ACM, pp. 25–36.

[27] Rødskog, D. Case Studies of Modeling Distributed Algorithms in ABS. Master's thesis, University of Oslo, 2014. http://urn.nb.no/URN:NBN:no-44701.

[28] Strom, R. E., and Yemini, S. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng. 12*, 1 (Jan. 1986), 157–171.

[29] Takeuchi, K., Honda, K., and Kubo, M. An interaction-based language and its typing system. In *PARLE '94: Parallel Architectures and Languages Europe* (1994), C. Halatsis, D. G. Maritsas, G. Philokyprou, and S. Theodoridis, Eds., vol. 817 of *Lecture Notes in Computer Science*, Springer, pp. 398–413.

[30] Weiss, B. *Deductive Verification of Object-Oriented Software*. PhD thesis, Karlsruhe Institute of Technology, 2010.